10:16:20 OCA PAD INITIATION - PROJECT HEADER INFORMATION 06/15/88

Active

Project #: G-36-631 Cost share #: Rev #: 0
Center # : R6506-0A0 Center shr #: OCA file #:
Work type : RES
Contract#: CCR-8806358 Mod #: Document : GRANT
Prime #: Contract entity: GTRC

Subprojects ? : N
Main project #:

Project unit: ICS Unit code: 02.010.142
Project director(s):
AHAMAD M ICS

Sponsor/division names: NATL SCIENCE FOUNDATION / GENERAL
Sponsor/division codes: 1000 / 7 000

Award period: 880601 (effective) 901130 (performance) 910228 (reports)

Sponsor amount	New this change	Total to date
Contract value	64,274.00	64,274.00
Funded	64,274.00	64,274.00
Cost sharing amount		54,166.00

Does subcontracting plan apply ?: N

Title: USING MULTICAST COMMUNICATION FOR RESOURCE FINDING IN DISTRIBUTED SYSTEMS

PROJECT ADMINISTRATION DATA

OCA contact: Steven K. Hett 894-4820

Sponsor technical contact

NATHANIEL MACON
(202)357-7375
NATIONAL SCIENCE FOUNDATION
CISE/CCR
WASHINGTON, D.C. 20550

Sponsor issuing office

SHARON GRAHAM
(202)357-9621
NATIONAL SCIENCE FOUNDATION
DGC/CISE
WASHINGTON, D.C. 20550

Security class (U,C,S,TS) : U ONR resident rep. is ACO (Y/N): N
Defense priority rating : N/A NSF supplemental sheet
Equipment title vests with: Sponsor GIT X
SUN 3/60 M-4-10

Administrative comments -
INITIATION.

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 08/16/91

Project No. C-36-651 Center No. R6506-0A0

Project Director AHAMAD M School/Lab COMPUTING

Sponsor NATL SCIENCE FOUNDATION/GENERAL

Contract/Grant No. CCR-8806358 Contract Entity GTRC

Prime Contract No.

Title USING MULTICAST COMMUNICATION FOR RESOURCE FINDING IN DISTRIBUTED SYSTEMS

Effective Completion Date 910531 (Performance) 910831 (Reports)

Closeout Actions Required:	Y/N	Date Submitted
Final Invoice or Copy of Final Invoice	N	
Final Report of Inventions and/or Subcontracts	Y	910814
Government Property Inventory & Related Certificate	Y	
Classified Material Certificate	N	
Release and Assignment	N	
Other	N	

Comments**FORMERLY G-36-651. INVOICING VIA NSF LINE OF CREDIT. 98A SATISFIES REQUIREMENT FOR PATENT REPORT.

Subproject Under Main Project No.

Continues Project No.

Distribution Required:

Project Director	Y
Administrative Network Representative	Y
GTRI Accounting/Grants and Contracts	Y
Procurement/Supply Services	Y
Research Property Managment	Y
Research Security Services	N
Reports Coordinator (OCA)	Y
GTRC	Y
Project File	Y
Other	N
	N

NOTE: Final Patent Questionnaire sent to PDPI.

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT (SUBPROJECTS)

Closeout Notice Date 08/16/91

Project No. C-36-651 Center No. R6506-0A0

Project Director AHAMAD M School/Lab COMPUTING

Sponsor NATL SCIENCE FOUNDATION/GENERAL

Project # C-36-616 PD AHAMAD M Unit 02.010.300 T
GRANT # CCR-8806358 MOD# NCE COMPUTING *
Ctr # 246R656-0A2 Main proj # C-36-651 OCA CO MSH
Sponsor-NATL SCIENCE FOUNDAT /GENERAL 107/000
USING MULTICAST COMM
Start 880615 End 910531 Funded 9,375.00 Contract 9,375.00

LEGEND
1. * indicates the project is a subproject.
2. I indicates the project is active and being updated.
3. A indicates the project is currently active.
4. T indicates the project has been terminated.
5. R indicates a terminated project that is being modified.

)MB Number 345-0058

NATIONAL SCIENCE FOUNDATION
800 G STREET, NW
WASHINGTON, DC 20550

PI/PD Name and Address

Mustaque Ahamad
School of Information & Computer Science
Georgia Tech Research Corp
Atlanta GA 30332

NATIONAL SCIENCE FOUNDATION
FINAL PROJECT REPORT

PART I - PROJECT IDENTIFICATION INFORMATION

1. Program Official/Org. Nathaniel Macon - CCR

2. Program Name SOFTWARE SYSTEMS PROGRAM

3. Award Dates (MM/YY) From: 06/88 To: 05/91

4. Institution and Address
Georgia Tech Research Corp
Administration Building
Atlanta GA 30332

5. Award Number 8806358

6. Project Title
Research Initiation: Using Multicast Communication for Resource Finding in Distributed Systems

This Packet Contains
NSF Form 98A
And 1 Return Envelope

PART IV — FINAL PROJECT REPORT — SUMMARY DATA ON PROJECT PERSONNEL

(To be submitted to cognizant Program Officer upon completion of project)

The data requested below are important for the development of a statistical profile on the personnel supported by Federal grants. The information on this part is solicited in response to Public Law 99-383 and 42 USC 1885C. All information provided will be treated as confidential and will be safeguarded in accordance with the provisions of the Privacy Act of 1974. You should submit a single copy of this part with each final project report. However, submission of the requested information is not mandatory and is not a precondition of future award(s). Check the "Decline to Provide Information" box below if you do not wish to provide the information.

Please enter the numbers of individuals supported under this grant.
Do not enter information for individuals working less than 40 hours in any calendar year.

	Senior Staff		Post-Doctorals		Graduate Students		Under-Graduates		Other Participants[1]	
	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.	Male	Fem.
A. Total, U.S. Citizens					1		2	1		
B. Total, Permanent Residents										
U.S. Citizens or Permanent Residents[2]:										
American Indian or Alaskan Native ...										
Asian					1					
Black, Not of Hispanic Origin							1			
Hispanic										
Pacific Islander										
White, Not of Hispanic Origin							1	1		
C. Total, Other Non-U.S. Citizens										
Specify Country 1.										
2.										
3.										
D. Total, All participants (A + B + C)					1		2	1		
Disabled[3]										

☐ Decline to Provide Information: Check box if you do not wish to provide this information (you are still required to return this page along with Parts I-III).

[1]Category includes, for example, college and precollege teachers, conference and workshop participants.

[2]Use the category that best describes the ethnic/racial status for all U.S. Citizens and Non-citizens with Permanent Residency. *(If more than one category applies, use the one category that most closely reflects the person's recognition in the community.)*

[3]A person having a physical or mental impairment that substantially limits one or more major life activities; who has a record of such impairment; or who is regarded as having such impairment. *(Disabled individuals also should be counted under the appropriate ethnic/racial group unless they are classified as "Other Non-U.S. Citizens.")*

AMERICAN INDIAN OR ALASKAN NATIVE: A person having origins in any of the original peoples of North America, and who maintain cultural identification through tribal affiliation or community recognition.

ASIAN: A person having origins in any of the original peoples of East Asia, Southeast Asia and the Indian subcontinent. This area includes, for example, China, India, Indonesia, Japan, Korea and Vietnam.

BLACK, NOT OF HISPANIC ORIGIN: A person having origins in any of the black racial groups of Africa.

HISPANIC: A person of Mexican, Puerto Rican, Cuban, Central or South American or other Spanish culture or origin, regardless of race.

PACIFIC ISLANDER: A person having origins in any of the original peoples of Hawaii; the U.S. Pacific Territories of Guam, American Samoa, or the Northern Marianas; the U.S. Trust Territory of Palau; the islands of Micronesia or Melanesia; or the Philippines.

WHITE, NOT OF HISPANIC ORIGIN: A person having origins in any of the original peoples of Europe, North Africa, or the Middle East.

THIS PART WILL BE PHYSICALLY SEPARATED FROM THE FINAL PROJECT REPORT AND USED AS A COMPUTER SOURCE DOCUMENT. DO NOT DUPLICATE IT ON THE REVERSE OF ANY OTHER PART OF THE FINAL REPORT.

Final Report for Project "Using Multicast Communication for Resource Finding in Distributed Systems" (CCR–8806358)

PART II – SUMMARY OF COMPLETED PROJECT

The goal of the project was to investigate distributed algorithms that can be used to find the locations of remote resources in a dynamic environment where resources can migrate between nodes. Such algorithms are necessary to allow sharing of resources between users of a distributed system. In studying the new and existing algorithms, not only the message cost of finding a resource was considered but the processing cost, which depends on the number of nodes that must participate in locating a resource, was also included. In bus based local area networks, a single broadcast message can reach all nodes but such a scheme has a very high processing overhead since all nodes need to process the request sent to find a resource. We have developed algorithms that use multicast communication and send the request to a small number of nodes. These algorithms provide significant savings in message and/or processing costs and have been applied to applications such as object invocation and load sharing. In store-and-forward networks, we have shown that very simple schemes that are easy to implement can have average message cost similar to several of the existing algorithms. Efficient schemes for implementing multicast communication have also been developed. Since several other problems in distributed systems can be modeled as instances of finding generalized resources, we also investigated how distributed mutual exclusion and replicated data management schemes can benefit from resource finding algorithms.

PART III – TECHNICAL INFORMATION

We have addressed the problem of resource finding in local area networks which provide support for multicast communication in the hardware and also in store-and-forward networks. The following is a brief summary of the results of research that was supported by award CCR–8806358.

1. *Using Multicast Communication for Resource Finding in Local Area Networks*

   If a resource is found using a broadcast message, every node in the distributed system must receive the request message each time a location operation is performed. We have developed a method that can reduce this overhead [2]. In this method the universe of resource names is partitioned into a relatively small number of groups and each group is assigned a unique address. Nodes storing the location of a resource belonging to a particular group instruct their network interfaces to receive messages sent to the group address. A node attempting to find a resource first determines the address of the group to which the resource belongs. This is accomplished via a well known hash function. A

multicast message is then sent to that address. Using analytic and simulation models, we investigated the performance of the scheme. Our conclusion is that even when there is a large number of resources in the system which migrate frequently, the multicast scheme is very efficient as a small number of nodes participate in each resource finding operation.

The multicast scheme sends a single location message (similar to broadcast) and hence the response time of the resource finding operation is small. It is possible to further reduce the number of nodes that need to handle request messages for finding a resource when increased response time is acceptable. We have investigated a generalized polling scheme [3] in which the request for a resource's location is successively sent to groups of nodes until the resource is located. In this case, the cost function is not only the number of nodes that process the request messages but also the time required to find the node where the resource resides, taking into account network contention and errors. Using a detailed network model, we have developed algorithms that can be used to partition the nodes in the system into multicast groups and determine the order in which the groups should be polled to minimize the cost.

The multicast facilities of a local area network can also be exploited to implement load sharing which is also an instance of the resource finding problem since the logical resource to be found is the node with the least load. We present a scheme in [6] which associates nodes having similar workloads with common multicast groups. The membership of the groups changes dynamically as the load varies at the nodes. When a new task arrives, a sequence of requests may be sent as multicast messages; starting with the multicast group consisting of idle nodes. A node is chosen to execute the new task from the ones that respond to the first multicast request. If some node is idle, it is found in a single message. Using a simulation study we showed that this scheme achieves close to optimal load sharing by reducing the processing overhead.

2. *Resource Finding in Store-and-Forward Networks*

In [1], we model the process of searching for a resource in a distributed system whose nodes are connected through a store–and–forward network. In this work, our main goal was to understand the message cost of resource finding in such networks. Based on this model, we show lower and upper bounds on the number of messages necessary to find a resource when nothing is known about the location of the resource. Although similar bounds have been derived for other methods but they require nodes to maintain additional information (nodes must store addresses in the forwarding addresses method).

The model is also used to establish results about the complexity of finding optimal algorithms to locate a resource when the probability distribution for the location of the resource in the network is known. We show that the optimization problem is NP-hard for general networks. Finally we show an algorithm for tree networks which can be specialized to polynomial algorithms for special kinds of trees. (The polynomial

algorithms can be used as the basis of heuristic algorithms for general networks.) An application of this algorithm yields optimal search algorithms for bidirectional ring networks.

3. *Efficient Message Delivery to Dynamic Multicast Groups*

   Dynamic multicast groups arise in many distributed applications. In particular, such groups can be used to reach a set of nodes that can locate a resource. We have addressed two separate problems in the area of message delivery to multicast groups [4,5]. First, to reduce the cost of message delivery to a group, we investigated the use of a spanning tree of the members of a multicast group. The problem of constructing a minimal spanning tree of the members of a multicast group which include only a subset of the nodes in the network is computationally intractable. In dynamic groups, the problem becomes more serious because the tree needs to be recomputed when the membership of the group changes. We developed two heuristic algorithms which update the tree incrementally as the membership changes. The goal is to reduce the total bandwidth required for sending data and control messages. The broadcast tree based algorithm makes use of a tree structure in the network, and the other algorithm joins a new member to the node that is nearest to it and is already in the multicast tree. A simulation model was used to study the performance of the algorithms. Although the cost of delivering data messages is higher in the proposed algorithms, they have lower overall cost when the group is dynamic and the cost of maintaining the trees is also included.

   We also investigated how reliable and ordered message delivery to dynamic multicast groups can be implemented. In point-to-point networks, we exploit the tree structure maintained for message delivery to ensure ordering. For local area networks, we developed an efficient protocol which makes use of the broadcast communication medium. The protocol ensures that not only members of a single group receive messages in the same order, but processes in different groups also receive common messages in the same order.

4. *Distributed Mutual Exclusion and Management of Replicated Data*

   The resource finding problem is quite general and efficient algorithms for it can be used in several other problems in distributed systems. For example, in distributed mutual exclusion, the privilege that allows a node to enter the critical section (CS) can be considered a logical resource which migrates as nodes make requests to enter the CS. To obtain access to the CS, the requesting node has to find the current holder of the privilege or another node which can transfer the privilege to it. We studied a communication efficient distributed mutual exclusion algorithm [7] based on the forwarding addresses method that was developed for finding resources. We developed a formal model of the algorithm's execution, which enabled us to prove its correctness. The formal model is also used to show that an execution history of the algorithm when concurrent requests are made (the normal case) is equivalent to a history in which the

requests are made serially. Based on this fact we proved a logarithmic upper bound on the average number of messages needed per access to the critical section.

In a replicated data system, the logical resource is the data which has copies at several nodes (this is done to increase the availability of the data). A transaction typically needs to locate a set of copies before it is allowed to access the data. Although the nodes where the copies are stored may be known, due to node failures and recoveries, the currently operational nodes with the data copies may not be known. In [10], multicast communication is used to find the set of nodes that can allow a transaction to read or update the data. The new protocol not only provides high data availability but also provides a high degree of load sharing between the nodes having copies of the data. We used a simulation model to demonstrate that the high degree of load sharing leads to significant improvement in transaction response time compared to existing replication management protocols. Our other work in this area has produced results that can be used to determine optimal vote and quorum assignments as well as a unified mechanism for modeling the operation of a general class of protocols [8,10,11,12,13]. We have also developed new schemes for building fault-tolerant distributed applications based on the replication of computations and checkpointing and rollback techniques [9,14,15].

In addition to the research discussed above, the award also supported our initial work in weakly consistent distributed shared memories [15,16,17]. These initial results were important in preparing a new proposal that has been funded by NSF this year. The dissertation work of several doctoral students was also partly supported by this award. Three of them have already completed their Ph.D. and two others are expected to complete in the next six months.

Publications from Research Funded by Award CCR-8806358

1. J. M. Bernabeu, M. Ahamad, and M. H. Ammar, *Resource Finding in Store-and-forward Networks*, to appear in *Acta Informatica.*

2. M. Ahamad, M. H. Ammar, J. M. Bernabeu and M. Y. Khalidi, *Using Multicast Communication to Locate Resources in a LAN-based Distributed System*, Proc. of the 13th IEEE Conference on Local Computer Networks, October, 1988.

3. J. M. Bernabeu, M. H. Ammar and M. Ahamad, *Optimal Selection of Multicast Groups for Resource Location in a Distributed System*, in Proc. of IEEE INFOCOM, April 1989.

4. N. Belkier and M. Ahamad, *Low Cost Algorithms for Message Delivery in Dynamic Multicast Groups*, in Proc. of Ninth International Conference on Distributed Computing, June 1989.

5. N. Belkier and M. Ahamad, *Ordered and Reliable Message Delivery in Dynamic Multicast Groups*, Georgia Tech Technical Report, April 1990. (submitted for publication).

6. M. Ahamad and N. Belkier, *Using Multicast Communication for Dynamic Load Balancing in Local Area Networks*, in 14th IEEE Conference on Local Computer Networks, October, 1989.

7. J. M. Bernabeu and M. Ahamad, *Applying a Path Compression Technique to Obtain an Efficient Distributed Mutual Exclusion Algorithm*, in Lecture Notes in Computer Science (Proc. of International Workshop on Distributed Algorithms, Nice, September 1989).

8. S. Y. Cheung, M. Ahamad and M. H. Ammar, *Multi-dimensional Voting*, to appear in *ACM Transactions on Computer Systems.*

9. M. Ahamad, P. Dasgupta and R. J. LeBlanc, *Fault-tolerant Atomic Computations in an Object-based Distributed System*, Distributed Computing Vol 4(2), 1990.

10. S. Y. Cheung, M. H. Ammar and M. Ahamad, *The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data*, Proc. of Sixth IEEE International Conference on Data Engineering, February, 1990 (To appear in *IEEE Transactions on Knowledge and Data Engineering*).

11. M. H. Ammar, M. Ahamad, S. Y. Cheung, *Performance of Quorum Consensus Protocols for Mutual Exclusion from the User's Point of View*, Proc. of the 2nd IEEE Workshop on Future Trends in Distributed Computing Systems, 1990. (submitted for journal publication).

12. M. Ahamad, M. H. Ammar, S. Y. Cheung, *Optimizing the Performance of Replica Control Protocols*, Proc. of the IEEE Workshop on the Management of Replicated Data, Houston, Texas, 1990.

13. S. Y. Cheung, M. Ahamad and M. H. Ammar, *Optimizing Vote and Quorum Assignments for Reading and Writing Replicated Data*, IEEE Trans. on Knowledge and Data Engineering, September, 1989 (also in Proc. of Fifth International Conference on Data Engineering, February, 1989).

14. L. Lin, M. Ahamad, *Checkpointing and Rollback-Recovery in Object-based Distributed Systems*, in Proc. of 20th Fault-tolerant Computing Systems Symposium, July 1990.

15. M. Ahamad and L. Lin, *Using Checkpoints to Localize the Effects of Faults in Distributed Systems*, in Proc. of IEEE Symposium on Reliable Distributed Systems, October, 1989 (submitted for journal publication).

16. P. W. Hutto and M. Ahamad, *Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories*, Proc. of the 10th International Conference on Distributed Computing Systems, Paris, France, 1990. (submitted for journal publication).

17. M. Ahamad, P. W. Hutto and R. John, *Implementing and Programming Causal Distributed Memory*, Proc. of the 11th International Conference on Distributed Computing, May 1991.

18. M. Ahamad, M. Chelliah, P. Dasgupta, R. LeBlanc and M. Pearson, *Shared Memory Programming in Distributed Systems*, Georgia Tech Technical Report GIT-CC-90/63. (submitted for publication).

Thesis Abstracts

Location Finding Algorithms for Distributed Systems

a Thesis
Presented to
The Faculty of the Division of Graduate Studies

By

José Manuel Bernabéu Aubán

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in the School of Information and Computer Science

Georgia Institute of Technology
December 1988

Summary

One of the problems encountered in distributed systems is how to find the location of the resources needed by a computation. In many situations the location may have to be found at run time, when the resource is accessed, thus the efficiency of the location algorithm will affect the performance of the system. In general, the larger the distributed system, the more the number of processors at which a resource may reside at the time it is accessed. The general problem of resource location in distributed systems has not been addressed adequately, and most of the systems have adopted ad hoc solutions without a careful study of the performance of the algorithms used. In this thesis it is studied the problem of finding the location of resources in order to get a better understanding of the factors affecting the cost of a location algorithm. This study will make it possible to judge proposed algorithms as well as to come up with new ones, optimized for particular systems.

Most distributed systems are based on bus networks that have broadcast and multicast capabilities. The thesis first describes an efficient location method that takes advantage of the multicast capabilities of these networks to reduce the computation cost of resource location finding. Performance results based on a simulation of the scheme are presented, showing that the method is a simple and efficient one. An approximate analysis is also presented, and it is shown that the analysis provides an extremely good approximation for low and high values of the load in the system. In another multicast scheme for broadcast networks, the thesis considers a system in which no references to resources are stored in the network except where the resource resides. Besides the CPU cost, response time costs are also considered, and a cost formula is found for the scheme. Based on this cost formula, an algorithm is presented to find an optimal sequence of multicast groups to be used in locating a resource.

The thesis then considers the communication costs incurred by location finding algorithms in store–and–forward networks. A model of such system is first constructed and, based on this model, a worst case analysis is performed to obtain a lower bound on the number of messages needed to locate a resource when no information about the location of the resource is available at the node conducting the search. It is also shown that when the searcher node has the probability distribution indicating the location of the resource in the system, the problem of finding the optimal way to traverse the network has only a polynomial time algorithm for restricted classes of networks.

The use of hint tables can reduce the cost of resource location when a resource is used repeatedly. The thesis presents a model of the usage of hint tables and shows how it affects the performance of finding the location of resources.

Optimizing the Performance of Quorum Consensus Replica Control Protocols

a Thesis
Presented to
The Faculty of the Division of Graduate Studies

By

Shun Yan Cheung

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in the College of Computing

Georgia Institute of Technology
July 1990

Summary

This thesis considers the performance of synchronziation protocols based on *quorum consensus* in distributed systems. In these protocols, an operation can proceed if permission can be obtained from nodes that constitute a *quorum group*. The collection of all quorum groups is a *quorum set*. Voting can be used to define quorum sets and it is appealing because it is flexible and can be easily implemented. However, voting cannot be used to represent all quorum sets.

We first study the problem of optimizing the system availability of quorum consensus methods and presents a direct method for finding the optimal quorum set for mutual exclusion, and reading and writing of replicated data. We show that the optimal system availability can be achieved by voting.

The thesis then considers optimizing an arbitrary performance measure. Changes in the quorum set cause performance changes in a discrete and highly complex manner, and a direct method is difficult to obtain. However, when the quorum set is given, the system behavior is fixed and the performance can then be computed with relative ease. The thesis presents an efficient algorithm for generating the universe of vote assignable quorum sets. The optimal voting parameter settings can be obtained by a search.

The thesis next presents a non-voting based quorum consensus protocol, called the Grid Protocol, that has small quorum groups. An analysis shows that the data availability of this protocol can be as high as voting and simulation results show that transactions using the grid protocol can have lower response time than voting.

Finally, the *multi-dimensional voting* concept is investigated where vote and quorum assignments are k-dimensional vectors of non-negative integers. Each dimension of the

vote and quorum assignment is similar to voting and the quorum requirements in different dimensions can be combined in a number of ways. Multi-dimensional voting is as general as quorum sets but has the advantage that it is flexible and easy to implement. Several replica control protocols are implemented using multi-dimensional voting which illustrate the versatility of this technique.

Localizing the Effects of Failures in Distributed Systems

Technical Report GIT-ICS-90/43
October 1990

Luke Lin
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280 USA

A Dissertation
Presented to
The Academic Faculty

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Information and Computer Science

Summary

This dissertation presents *checkpointing and rollback-recovery* techniques for building fault tolerant distributed systems. Checkpoints are used to ensure that computations are able to make *forward progress* in the presence of failures, which is essential when computations are long running. The concept of *locality* is introduced, which is the objective of restricting a checkpoint or rollback operation to a single computation to contain the effects of failures. Since the number of failures increases as a system becomes larger, an uncontrolled propagation of a rollback each time a failure occurs can impede forward progress when locality is not satisfied. In addition, we require that programmers be able to control the frequency of checkpoints in a computation, independent of other computations; thus, making it possible to provide different degrees of fault tolerance for individual computations. Algorithms that achieve locality are presented for distributed message based systems, distributed object based systems, and replicated computations.

We first address locality with respect to distributed message based systems. When computations are deterministic, we develop a solution that is based on a combination of consistent and pessimistic checkpointing techniques. When computations may be non-deterministic, we show how the blocking required to enforce locality can be reduced to enhance concurrency. Next, we present checkpoint and rollback algorithms for distributed object based systems. By exploiting the structure of objects and operation invocations, we derive algorithms that involve fewer participants compared to when invocations are

treated as messages and approaches from message based systems are used. These results are then extended to satisfy the locality of checkpoint and rollback operations in object based systems. Finally, we discuss how checkpointing can be integrated with replicated computations. We develop an approach that retains the attractive features from both checkpointing and replication while potentially reducing the cost of achieving fault tolerance. In many situations, a checkpointed replicated computation can potentially outperform a checkpointed non-replicated computation.

Cover Pages of Publications

To appear in Acta Informatica.

Resource Finding in Store–and–Forward Networks*

José M. Bernabéu–Aubán
Mustaque Ahamad
Mostafa H. Ammar

Abstract

We present a model of searching for a resource in a distributed system whose nodes are connected through a store–and–forward network. Based on this model, we show a lower bound on the number of messages needed to find a resource when nothing is known about the nodes that have the current location of the resource. The model also helps us to establish results about the time complexity of determining a message optimal resource finding algorithm when the probability distribution for the location of the resource in the network is known. We show that the optimization problem is NP-hard for general networks. Finally we show that optimal resource finding algorithms can be determined in polynomial time for a class of tree networks and bidirectional rings. The polynomial algorithms can be used as a basis of heuristic algorithms for general networks.

*This work was supported in part by NSF grants CCR–8806358 and NCR–8604850.

Using Multicast Communication to Locate Resources in a LAN-Based Distributed System†

Mustaque Ahamad *Mostafa H. Ammar* *José M. Bernabéu-Aubán* *M. Yousef Khalidi*

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

In this paper we present a resource (e.g., file, process) location scheme which exploits the multicast communication capability of local area networks. In the scheme, the universe of resource names is partitioned into a relatively small number of groups and each group is assigned a unique address. Nodes storing the locations of resources belonging to a particular group instruct their network interfaces to receive all location messages sent to the group address. To locate a resource, a node first determines the address of the group to which the resource belongs (this can be accomplished via a well-known hash function), and a multicast message is then sent to the address. The algorithm performance is studied by means of simulation, and approximate closed form solutions are derived for systems operating at heavy and low loads. The scheme's performance is compared with that of broadcast, and it is shown that the proposed scheme performs much better than broadcast alone.

Introduction

e advantages offered by distributed systems include resource ıring, fault-tolerance and parallel execution of a computation. e programming of distributed systems is more complex than ıtralized ones due to the unavailability of the global state of : system. For example, in a dynamic system where resources g., files, processes) can be migrated between nodes, a user st program an algorithm to find the current location of a reırce needed by his or her computation. This can be avoided if rs are provided with the abstraction of a unified system where : location of resources is transparent to them. Resources are erred to by names and, at runtime, the system determines the rent location of a named resource.

Many schemes have been proposed for finding the location a named resource. Conceptually, there exists a database that res the associations between resource names and their locaıs. This database can be partitioned and stored at one or re nodes that are called *name servers*. When a remote reırce, R, needs to be accessed, the request for its location uld be sent to a name server that stores R's location. The system must also implement algorithms to update the information stored by the name servers when resources are created or deleted or when they are migrated. To avoid this, the database can be distributed in such a way that a name server at a node maintains a list of only resources local to the node. In such a system a remote resource can be located by broadcasting its name, and having the node where the resource is located respond. This scheme is used in Clouds [DLS85] for locating remote objects. Broadcast can also be used when other schemes fail to locate a resource.

We are concerned with a distributed system that uses a broadcast bus local area network. In such an environment, all network interfaces receive every message carried on the bus. A particular message is delivered to the attached node only if it is sent to a destination address that the interface has been instructed to recognize. Such addresses will at least include the broadcast address and the node's own address. Thus, if a broadcast message is used to locate a resource, the message will be delivered to all the nodes in the distributed system. This in turn will cause all the nodes to search their local resource directories which represents a wastage of CPU time at all nodes except the one where the resource resides.

In this paper, we explore the design of a distributed name server where multicast communication is used to locate the requested resource. In such a system, a particular message sent to locate a resource will be delivered to only a subset of the nodes in the system. The availability of bus interface communications technology that supports multicast in the hardware provides the motivation for this work. Our goal is to design a location scheme that is simple from the point of view of a node that needs to find a resource but, at the same time, reduces the number of nodes that must participate in the location process.

We associate a multicast address with each resource name and this address is used to communicate with the name server of the resource. Each node receives messages sent to multicast addresses corresponding to the resources whose locations are stored by the local name server. Typically, a limited number of multicast addresses will be available at each interface for use by the resource location operations. Since the number of resources in the distributed system can be large, the resource name to multicast address mapping is many-to-one. For such a system, we present the algorithms to be executed when a resource is created, deleted or a request is made for finding its location. We also study the performance of the multicast scheme and compare it with broadcast. The cost measure used is the number of nodes that process messages sent for finding a resource or for updat-

†This work has been partially supported by NSF grants CCR-8806358, R-8604850, and CCR-8619886.

Optimal Selection of Multicast Groups for Resource Location in a Distributed System[†]

José M. Bernabéu-Aubán *Mostafa H. Ammar* *Mustaque Ahamad*

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

In this paper we present a protocol to locate (or find) named resources in a distributed system which uses the multicast capabilities of the underlying network. Each node in the network uses a sequence of node groups, and each node group is associated with a unique multicast address. To locate a resource, the searching node sequentially polls each one of the groups until the resource is found. This scheme is a generalization of both pure polling and broadcast. Our basic aim is to show how to obtain an optimal division of the nodes into multicast groups. To that end, the protocol is analyzed and an efficient algorithm is given that provides a group division minimizing the expected cost per location operation.

1 Introduction

Distributed systems offer many advantages over centralized ones, including fault-tolerance, resource sharing and increased parallelism. The task of programming a distributed system, however, is more difficult because the global system state is not available. In particular, in systems where resources (e.g., files, processes) can be migrated between nodes, it would be necessary for the user to implement a procedure to find resources needed by the computation. To avoid this, the system must offer the users the abstraction of a unified system in which the location of resources is transparent to them. In such a system, the Operating System should implement an algorithm to find the location of a remote resource. When a resource may be used repeatedly at a node, caching its address locally [1,2,3,4], is a widely used technique for reducing the cost of determining the location of a resource. Since the cache information may be incorrect, the general problem of finding the resource still exists when caching is used.

A widely used scheme to find resources involves the use of name servers [1,5]. In its simplest form, one of the nodes in the network is designated as the name server for the whole system. When a node needs to locate a resource, it directs a request to the name server. When a resource moves between nodes, an update message is sent to notify the name server. In a large system, such a name server would become a bottleneck, degrading the performance of the system. Also, the single name server approach would be especially vulnerable to node failures. A more general approach can distribute the name server task among several nodes and, a particular name server usually takes care of only a part of the resource name space. The problem now is to decide which name server to contact to find the location of a resource. A resource's location may now be found by broadcasting (actually multicasting [6,7]) to all name servers requesting that they provide the resource's address. Another approach, used in the R* system [2], is to encode the name of the node where a resource was created in the resource's name. Then that node will function as the resource's name server.

In the absence of name servers, a node wishing to determine the location of a resource, can send a broadcast message to all nodes and make them search their local directories. This is the approach taken in the *Clouds* operating system [8]. Broadcasting, though simple, would waste computational resources at every node, where it would compete with the local computations for CPU time. For large rates of location requests this would rapidly degrade the performance of the entire system.

At the other extreme, if the individual nodes are polled sequentially, this would certainly decrease the amount of CPU time wasted in the system (especially if the nodes more likely to know about the resource were consulted first). However this approach would also increase the bandwidth utilization, because many messages will be sent. Since the messages are sent sequentially, the real disadvantage of this approach is that the location operations would take longer (larger response time).

In this paper we present a location protocol which considers a cost measure that includes both the CPU utilization and the response time. The approach taken is based on a scheme in which the nodes in the network are divided into disjoint multicast groups, and are polled by a sequence of multicast messages. The two approaches mentioned above are just special cases when all nodes are reached by a single message (broadcast) or when only one node is reached with each message (polling). We also present a cost model for the system and an efficient algorithm which, based on the probability distribution of a resource's location among the nodes in the network, finds the optimal decomposition into disjoint groups, as well as the optimal sequence in which the groups should be polled.

In section 2 we give a description of the protocol operation. In section 3 we present the model of the system to be used for the cost analysis carried out in section 4. Section 5 describes an algorithm to determine an optimal multicast grouping. Some numerical examples are presented in section 6. In section 7 we

[†]This work was supported in part by NSF grants NCR-8604850 and CCR-8806358.

Low Cost Algorithms for Message Delivery in Dynamic Multicast Groups[†]

Nasr E. Belkeir *Mustaque Ahamad*

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

abstract

Dynamic multicast groups arise in many distributed applications. A spanning tree of the members of a multicast group can be used to reduce the cost of message delivery to the members. The problem of constructing a minimal spanning tree of the members of a multicast group which includes only a subset of the nodes in the network is computationally intractable. In dynamic groups, the problem becomes more serious because the tree needs to be recomputed when the membership of the group changes. We develop two heuristic algorithms which update the tree incrementally as the membership changes and reduce the total bandwidth required for sending data and control messages. The broadcast tree based algorithm makes use of a tree structure in the network, and the other algorithm joins a new member to the node that is nearest to it and is already in the multicast tree. A simulation model is used to study the performance of the algorithms.

1 Introduction

Multicast communication allows a message to be addressed to any subset of nodes in a network. In many distributed applications, a set of nodes need to be queried or notified (e.g., name servers) when a certain event happens. A multicast facility can not only simplify the implementation of such applications but can also deliver the message to the destination nodes more efficiently compared to when a separate copy of the message is sent to each destination node. We define a *multicast group* as a set of nodes to which a message needs to be delivered. Although a multicast group is sometimes defined by a set of processes, we consider the nodes where these processes execute because the cost of message delivery in this paper depends only on the nodes in the group.

The membership of a multicast group changes with time in many applications. For example, when data is replicated to enhance its availability, the nodes that store copies of a particular data item define a multicast group. When a node storing a copy fails, it can be deleted from the group, and other nodes can be added to the group to increase the availability of the data. In [2], a multicast based resource finding scheme is proposed in which the set of nodes that receive a message sent to a particular address changes as resources are created, deleted or migrated. Other distributed algorithms that can make use of multicast communication with dynamic membership include commit and checkpointing algorithms [14], and load balancing [17]. One of the emerging application of distributed systems is multimedia teleconferencing which naturally defines a dynamic communication group since the set of participants varies with time. Thus, dynamic multicast groups arise both at the application level as well as in the operating system.

A multicast message can be delivered by sending a copy of the message to each member of the group. However, this wastes bandwidth because multiple copies of a message may be sent over the same communication link. This can be avoided by constructing a spanning tree of the members of a multicast group and then delivering the message by forwarding copies of it along the edges of the tree. However, unlike a broadcast spanning tree [8,11], the problem of constructing a minimum multicast spanning tree is computationally intractable (NP-complete) since it is an instance of the steiner tree problem [12]. Thus, for large networks, construction of an optimal multicast tree is infeasible. When the membership of the group is dynamic, the problem becomes more serious because the tree needs to be recomputed each time a member deletes from the group or a new member is added to it.

In this paper, we will investigate heuristics for constructing low cost multicast spanning trees in a dynamic environment. More precisely, we will present schemes that incrementally update the multicast tree as the membership of the group changes. We will evaluate the performance of these algorithms by means of simulation and compare them with the best known heuristic algorithm developed by Wall [18] which computes the multicast tree structure for a given membership of the group.

2 Related Work

Wall [18] investigated techniques for organizing tree structures that can be used to efficiently deliver a multicast message to its destination nodes in a point-to-point network. He described a distributed algorithm which takes the set of nodes in the group as input and produces a low cost spanning tree that contains these nodes. In a later section, we will consider the details of this algorithm because it will be used to evaluate the effectiveness of the algorithms proposed in this paper.

The problem of routing a message to multiple destinations has been addressed by many researchers. In [1], an extension to the DOD internet protocol [16] is proposed which allows multiple destination addresses in a packet. At gateway nodes, the packet can be replicated and sent along different branches depending on the addresses contained in the packet header. In [9], many schemes have been explored for routing of multicast messages in an internetwork and extended LANs. The construction of a spanning tree for a multicast group in a bus-based hypercube is described in [15]. A performance study of several routing

[†]This work was supported in part by NSF grant CCR-8806358.

Ordered and Reliable Message Delivery in Dynamic Multicast Groups*

Nasr E. Belkeir
Mustaque Ahamad

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
(404)-894-2593

Abstract

Many distributed applications can be implemented efficiently using multicast communication which allows a message to be sent to a group of processes. Dynamic process groups arise naturally in distributed environments. In such groups, processes can join or leave the group asynchronously. In this paper, we address the problem of ordered and reliable message delivery to dynamic multicast groups in distributed systems. We present an efficient protocol which makes use of the broadcast communication medium. The protocol ensures that not only members of a single group receive messages in the same order, but processes in different groups also receive common messages in the same order. We also discuss how the protocol can tolerate node and communication failures.

*This work was supported in part by NSF grants CCR-8806358 and CCR-8619886.

Appeared in Proc. of IEEE Conference on Local Computer Networks, October 1989.

Using Multicast Communication for Dynamic Load Balancing in Local Area Networks *

Mustaque Ahamad *Nasr E. Belkeir*
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

In local area networks, a multicast message can be delivered to a subset of hosts in the network. Such a communication facility can be exploited to efficiently implement distributed applications. We propose a multicast based algorithm for load balancing in a local computer network. The ability of a host to determine its membership to a multicast group based on its workload makes it possible to send a transfer request to only those hosts which have lower workload than the host that wants to transfer a task. This reduces the overhead of load balancing because a host does not waste computational resources when it cannot accept a task. We use a simulation model to investigate the performance of the multicast based algorithm and compare it to algorithms that use unicast (point-to-point) communication.

1 Introduction

Multicast communication allows a message to be addressed to any subset of hosts in a network. A multicast facility can not only simplify the implementation of many distributed applications but it can also exploit the capabilities of current local area network (LAN) interfaces that provide hardware support for multicast address recognition [4,15]. We define a *multicast group* associated with address m as the set of hosts that receive a message sent to m. Current network interfaces allow a host to join and delete from a multicast group by instructing its network interface to start/stop accepting messages sent to a particular address.

In a local computer network, many hosts (workstations) are frequently idle while others are heavily loaded. *Load balancing* can be used in such a system to allow the sharing of computational resources by transparently distributing the system workload among the hosts in the network. The two main components of a load balancing algorithm are the *transfer* policy, which determines whether to process a task locally or remotely, and the *placement* policy, which determines to which host a task selected for transfer should be sent. When these policies react to the system state they are called *dynamic* policies. Hosts need to exchange information about their workload for implementing dynamic load balancing. In this paper we investigate efficient dynamic load balancing algorithms that use multicast communication to determine the host where an incoming task should be sent.

The load balancing algorithm is based on a simple scheme in which the current workload at a host determines its membership to a particular multicast group. Hosts belonging to a group have similar workloads. When a host h wants to explore if it is possible to transfer an incoming task, it queries hosts with lower workloads by sending multicast messages to groups that correspond to loads lighter than the current workload of h. Thus, a host can communicate with a group of hosts in a particular workload range without collecting information about the workload at various hosts. This is made possible by the ability of each host to change its membership based on its workload. Since multicast messages are sent to only groups corresponding to lower workloads, hosts with higher workload do not waste computational resources processing requests from hosts that have lower workload. This can significantly reduce the overhead of load balancing. Also, instead of probing a set of hosts to determine one with the lowest load, the algorithm only needs to send messages until the first response is received. Since each host can determine its membership to a group locally, no message overhead is incurred for maintaining the groups.

In this paper, we present the details of a multicast based load balancing algorithm and show how it can be implemented in a LAN. We study the performance of the algorithm and compare it with other algorithms that have been proposed. In section 2, we describe the related work in the areas of multicast communication and load balancing. The system model is presented in section 3. We present the algorithm in section 4 and address the performance issues in section 5. Concluding remarks are described in section 6.

*This work was supported in part by NSF grant CCR-8806358.

Applying a Path–Compression Technique to Obtain an Efficient Distributed Mutual Exclusion Algorithm*

José M. Bernabéu–Aubán

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Apartado 22012, 46020 VALENCIA (Spain)

Mustaque Ahamad

School of Information and Computer Science
Georgia Institute of Technology
ATLANTA, GA 30332–0280 (USA)

Abstract

In this paper we present a distributed algorithm for mutual exclusion. The algorithm maintains a dynamic forest structure in which the paths between nodes are compressed as a result of requesting the Critical Section. We develop a formal model of the algorithm's execution, which enables us to prove its correctnes. The formal model is also used to show that an execution history of the algorithm when concurrent requests are made (the ususal case) is equivalent to a history in which the requests are made serially. Based on this fact we are able to prove a logarithmic upper bound on the average number of messages needed per critical section grant.

1 Introduction

Distributed systems offer many advantages including sharing of resources by processes executing at different nodes. In many applications, a process needs to obtain mutual exclusion before it can use a resource. We address the problem of designing an efficient distributed algorithm that can be used to achieve mutual exclusion in a distributed system.

A number of distributed mutual exclusion algorithms have been proposed [1]. The operation of many of the algorithms can be characterized by an information structure [2] that defines a set of processes that must be informed before acquiring the Critical Section (CS), and another set must be informed when the process releases the CS. Examples of these algorithms include [3,4,5,6,7,8]. The communication cost of all the algorithms except [5] is $O(N)$ where N is the number of nodes that share access to a resource. The algorithm described in [5] reduces the communication cost to $O(\sqrt{N})$ by imposing a logical structure on the processes.

Recently tree-based algorithms have been proposed for achieving mutual exclusion that require smaller number of messages [9,10,11]. However, in these algorithms some nodes need to

*This work was supported in part by NSF grants CCR–8806358 and CCR–8619886.

To appear in ACM Transactions on Computer Systems.

Multi-Dimensional Voting*

Mustaque Ahamad†
Mostafa H. Ammar†
Shun Yan Cheung‡

†College of Computing
Georgia Institute of Technology, Atlanta, GA 30332

‡Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322

Abstract

We introduce a new concept, *multi-dimensional voting*, in which the vote and quorum assignments are k-dimensional vectors of non-negative integers and each dimension is independent of the others. Multi-dimensional voting is more powerful than traditional weighted voting because it is equivalent to the general method for achieving synchronization in distributed systems which is based on sets of groups of nodes (quorum sets). We describe an efficient algorithm for finding a multi-dimensional vote assignment for any given quorum set and show examples of its use. We demonstrate the versatility of multi-dimensional voting by using it to implement mutual exclusion in fault-tolerant distributed systems, and protocols for synchronizing access to fully and partially replicated data. These protocols cannot be implemented by traditional weighted voting. Also, the protocols based on multi-dimensional voting are easier to implement and/or provide greater flexibility than existing protocols for the same purpose. Finally, we present a generalization of the multi-dimensional voting scheme, called *nested multi-dimensional voting*, that can facilitate implementation of replica control protocols that use structured quorum sets.

*This work was supported in part by NSF grants NCR-8604850 and CCR-8806358, and by the University Research Committee of Emory University.

Distributed Computing (1990) 4:69–80

DISTRIBUTED COMPUTING

Fault-tolerant atomic computations in an object-based distributed system *

Mustaque Ahamad, Partha Dasgupta, and Richard J. LeBlanc, Jr.

School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA

Received December 22, 1988/Accepted February 20, 1990

Mustaque Ahamad received his B.E. (Hons.) degree in Electrical Engineering from the Birla Institute of Technology and Science, Pilani, India. He obtained his M.S. and Ph.D. degrees in Computer Science from the State University of New York at Stony Brook in 1983 and 1985 respectively. Since September 1985, he is an Assistant Professor in the School of Information and Computer Science at the Georgia Institute of Technology, Atlanta. His research interests include distributed operating systems, distributed algorithms, fault-tolerant systems and performance evaluation.

Partha Dasgupta is an Assistant Professor at Georgia Tech since 1984. He has a Ph.D. in Computer Science from the State University of New York at Stony Brook. He is the technical project director of the Clouds distributed operating systems project, as well as a co-principal investigator of Georgia Tech's NSF-CER award. His research interests include building distributed operating systems, distributed algorithms, fault-tolerant systems and distributed programming support.

Richard J. LeBlanc, Jr. received the B.S. degree in physics from Louisiana State University in 1972 and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin – Madison in 1974 and 1977, respectively. He is currently a Professor in the School of Information and Computer Science of the Georgia Institute of Technology. His research interests include programming language design and implementation, programming environments, and software engineering. Dr. LeBlanc's current research work involves application of these interests in distributed processing systems. As co-director of the Clouds Project, he is studying language concepts and software engineering methodology for utilizing a highly reliable, object-based distributed system. He is also interested in specification-based software development methodologies and tools. Dr. LeBlanc is a member of the Association for Computing Machinery, the IEEE Computer Society and Sigma Xi.

Abstract. A distributed system can support fault-tolerant applications by replicating data and computation at nodes that have independent failure modes. We present a scheme called parallel execution threads (PET) which can be used to implement fault-tolerant computations in an object-based distributed system. In a system that replicates objects, the PET scheme can be used to replicate a computation by creating a number of parallel threads which execute with different replicas of the invoked objects. A computation can be completed successfully if at least one thread does not encounter any failed nodes and its completion preserves the consistency of the objects. The PET scheme can tolerate failures that occur during the execution of the computation as long as all threads are not affected by the failures. We present the algorithms required to implement the PET scheme and also address some performance issues.

Key words: Fault-tolerant computing – Atomicity – Distributed systems and replication

* This work was supported in part by NSF grants CCR-8619886 and CCR-8806358, and RADC contract number F30602-86-C-0032

Offprint requests to: M. Ahamad

To appear in IEEE Transactions on Knowledge and Data Engineering. An earlier version appeared in the Proc. of the International Conf. on Data Engineering, 1990.

The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data*

Shun Yan Cheung†
Mostafa H. Ammar‡
Mustaque Ahamad‡

†Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322

†College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

We present a new protocol for maintaining replicated data that can provide both high data availability and low response time. In the protocol, the nodes are organized in a logical grid. Existing protocols are designed primarily to achieve high availability by updating a large fraction of the copies which provides some (although not significant) load sharing. In the new protocol, transaction processing is shared effectively among nodes storing copies of the data and both the response time experienced by transactions and the system throughput are improved significantly. We present an analysis of the availability of the new protocol and use simulation to study the effect of load sharing on the response time of transactions. We also compare the new protocol with a voting based scheme.

*This work was supported in part by NSF grants NCR-8604850 and CCR-8806358, and by the University Research Committee of Emory University.

Optimizing Vote and Quorum Assignments for Reading and Writing Replicated Data

SHUN YAN CHEUNG, MUSTAQUE AHAMAD, AND MOSTAFA H. AMMAR, MEMBER, IEEE

Abstract—In the weighted voting protocol which is used to maintain consistency of replicated data, the availability of the data to read d write operations not only depends on the availability of the nodes ring the data but also on the vote and quorum assignments used. consider the problem of determining the vote and quorum assignnts that yield the best performance in a distributed system where le availabilities can be different and the mix of the read and write rations is arbitrary. The optimal vote and quorum assignments de- d not only on the system parameters such as node availability and ration mix, but also on the performance measure. We present an meration algorithm that can be used to find the vote and quorum ignments that need to be considered for achieving optimal perfornce. When the performance measure is data availability, an analyt- method is derived to evaluate it for any vote and quorum assignnt. This method and the enumeration algorithm are used to find the imal vote and quorum assignment for several systems. The enuration algorithm can also be used to obtain the optimal performance n other measures are considered.

ndex Terms—Availability, data replication, fault tolerance, replica trol methods, vote and quorum assignment, weighted voting.

I. INTRODUCTION

DISTRIBUTED system consists of a number of potentially unreliable nodes interconnected via a comnication subnetwork. The resources stored at the nodes be shared and when a node fails, the resources stored he node become unavailable. Replicating resources at erent nodes with independent failure modes can ence availability and fault tolerance, since a resource ld be available even when some nodes have failed. en data are replicated, care must be taken to preserve sistency among the various copies or *replicas*. In adon to increased availability, replication can also pro- improved performance of read transactions by reduc- the network communication cost since these sactions can access the data from the local replica. large number of replica control protocols have been eloped to maintain the consistency of replicated data In this paper, we address the issue of optimization for ting-based replica control protocol by deriving a genmethod for finding the optimal settings for the params of the protocol. We consider the voting mechanism because it has proven to be flexible and relatively easy to implement.

Voting has been used for various applications in distributed systems. In [2], Gifford proposed its use for synchronizing read and write operations on replicated files. Each file replica is assigned some number of votes and each operation is required to obtain a predefined quorum of votes to proceed. To ensure that a read operation returns the value installed by the last write operation, the read and write operations must acquire r and w number of votes, respectively, such that $r + w > L$, where L is the total number of votes assigned to all replicas. The values r and w are called the read and write quorum. Generally, $r + w = L + 1$ is used which ensures that each read quorum has a nonempty intersection with each write quorum. Since all replicas need not be updated when a write operation completes, timestamps or version numbers must be used in order to determine the value that is written most recently. When version numbers are used, each write quorum must also intersect with every other write quorum, i.e., $2w > L$ [2].

A number of replica control protocols have been derived from weighted voting. Eager and Sevcik introduced a dynamic scheme based on voting that allows the system to switch between normal and failure modes [3] (which have different values for read and write quorums). The system can also change the quorum assignment in the schemes presented in [4]–[6] and the vote assignment can be changed in the scheme described in [7]. Other protocols based on voting are presented in [8]–[10].

The problem of assigning votes to achieve mutual exclusion is addressed by Garcia-Molina and Barbara in [11]. When the quorum for each operation is a majority of all votes assigned, each operation will have mutually exclusive access to the data. In general, mutual exclusion can be guaranteed by defining a set of groups of nodes [12], called a coterie, such that any two groups in a coterie have a nonempty intersection. When voting is used, the groups of nodes that have a majority of the votes constitute a coterie (there exist coteries that cannot be obtained from any vote assignment [11]). In [11], it is shown that only a finite set of vote assignments need to be considered to get all coteries that can be obtained from vote assignments. Thus, it is not necessary to deal with the unbounded set of possible vote assignments. In another work, the same authors have considered the problem of

anuscript received September 28, 1988; revised July 11, 1989. This was supported in part by the National Science Foundation under s CCR-8806358 and NCR-8604850.
e authors are with the School of Information and Computer Science, gia Institute of Technology, Atlanta, GA 30332.
EE Log Number 8930638.

Optimizing the Performance of Quorum Consensus Replica Control Protocols*

Mustaque Ahamad
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Mostafa H. Ammar
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Shun Yan Cheung†
Dept. of Math. & Comp. Science
Emory University
Atlanta, Georgia 30322

Abstract

We present in this paper a summary of the results of our research in replica control protocols that are based on quorum consensus. In quorum consensus methods, operations are required to obtain permission from a quorum group of nodes to proceed to completion and the collection of quorum groups is called a quorum set. In the summary we present the techniques that we have developed for finding the quorum set that maximizes a given performance measure. We also present a brief discussion of the optimality of voting, a replica control protocol that can effectively reduce response time through load sharing, and the *multi-dimensional voting* (MD) technique, that can be used to define all quorum sets. An MD-voting based implementation of a dynamic quorum consensus protocol that allows the synchronization procedure to adapt to the current state of the system is also presented.

1 Introduction

Distributed systems offer many advantages including fault-tolerance which can be achieved by replicating resources at nodes with independent failure modes. When data (e.g., files) is replicated, algorithms must be used to maintain the consistency of the copies or *replicas* of the data. Such algorithms, called *replica control protocols*, implement rules for accessing the replicas to ensure correctness (e.g., single-copy serializability). A large number of replica control protocols have been proposed in the literature. These include voting, available copies, primary copy and many others. The main focus of these protocols has been to enhance *availability* by tolerating as many node and communication failures as possible. Availability can be defined as the steady state probability that a transaction is able to access the data successfully when it arrives to the system.

Data replication can also be used to improve other performance measures. For example, the execution of a transaction requires reading of data from disk, processing and possibly writing the data to the disk (when it is modified). If data is not replicated, all transactions that access data stored at a node must wait for the data to be read or written. When the data is replicated, load generated by the requests can be shared by nodes having the replicas and hence the response time of the transactions can be improved. Notice that the degree of sharing depends on the replica control protocol used. If read transactions can access any replica (a write transaction must update all replicas to ensure correctness), the load generated at each node by the read transactions will be $1/n$ compared to when no replication is used (n is the number of replicas).

We consider protocols that are based on quorum consensus [1]. An operation proceeds to completion only if it can obtain permission from nodes that constitute a *quorum group* [2]. Quorum groups used by conflicting operations have non-empty intersections to guarantee proper synchronization. The collection of quorum groups used by an operation is known as a *quorum set*. If each group in the quorum set intersects with every other group in the set, it is called a *coterie* [3] and it can be used to achieve mutual exclusion. Weighted voting [4] is a representation technique to define quorum sets so that quorum groups need not be listed explicitly. It is shown in [3] that there exist quorum sets that cannot be defined by voting.

We will summarize the results of our research in Sections 2–6 and they include techniques for finding the quorum set that optimizes a given performance measure, a replica control protocol for reducing response time, the multi-dimensional voting concept and an implementation of a dynamic replica control method.

2 Optimal System Availability

2.1 Homogeneous Systems

We have explored how optimal vote and quorum assignments can be obtained for a system for read and write transactions when their mix could be arbitrary. In [5], we considered the problem in a system where node reliabilities are identical. The performance measures considered are the *system* availability (i.e., the probability that some part of the system is available) to transactions without blocking (a transaction aborts instantaneously when the currently operational nodes do not have sufficient votes to form a desired quorum) and the mean response time when transactions wait for nodes to recover from failures until a quorum is available. One of the interesting results shows

*This work was supported in part by NSF grants NCR-8604850 nd CCR-8806358.

†Work was done while this author was at the College of Comput-g, Georgia Institute of Technology, Atlanta, Georgia.

To Appear in Proceedings of the 2nd Workshop on Future Trends in Distributed Computing

Performance of Quorum Consensus Protocols for Mutual Exclusion from the User's Point of View*

Mostafa H. Ammar *Mustaque Ahamad* *Shun Yan Cheung*
College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332

Abstract

A general class of protocols used for achieving mutual exclusion in distributed systems is quorum consensus. In these methods, an operation must obtain permission from a group of coordinators before it can proceed to completion. We consider a store-and-forward network with coordinators resident in some of the switching nodes. The main motivation for having multiple coordinators is to enhance system availability. Most studies of the availability of quorum consensus protocols have been concerned with assessing the system availability. In this work we consider the user point-of-view availability defined as the probability that a mutual exclusion operation originating at a given site can proceed to completion. We consider scenarios where the network links, as well as the network nodes may fail. Our objective is to analyze the user experienced availability and to determine how to best design a system so as to obtain high availability.

1 Introduction

A distributed system consists of a number of cooperating nodes interconnected by a communication network. The nodes communicate with each other through messages sent over the network. The advent of high speed networking allows for the possibility of running a multitude of new distributed applications. A number of these applications require mutually exclusive access to resources, for example, updates to a file must be synchronized. Synchronization methods used in distributed systems must be tolerant to node and network failures.

A general class of protocols used for achieving mutual exclusion in distributed systems is *quorum consensus*. In these methods, an operation must obtain permission from a group of *coordinators*, before it can proceed to completion. The groups that can grant permission must intersect with each other and a coordinator grants permission to only one operation at a time. This ensures that no two operations can proceed simultaneously. A set of groups, known as a *coterie* [1, 2], can be defined whose members are groups of coordinators that have the non-empty intersection (i.e., contain at least one common coordinator) property. In addition, if a group is a member of the coterie then it cannot be a subset of any other group in the coterie. The best known quorum consensus protocol is majority consensus [3] where each group consists of a majority of the coordinators. *Weighted Voting* [4] is a simple technique that can be used to implement quorum consensus protocols, where each node is assigned a number of votes and an operation must obtain a majority of votes before it can proceed to completion.

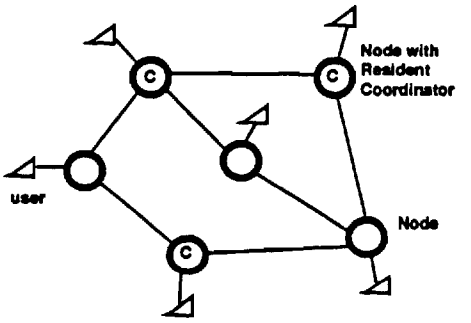


Figure 1: Coordinators, Users and Switching Nodes

Each assignment of votes uniquely defines a coterie. For example, in a system with four coordinators, A, B, C and D that are assigned 2, 3, 1, and 1 votes respectively, an operation requires at least four (a majority) votes to proceed. The coterie describing this is {{A,B}, {A,C,D}, {B,C}, {B,D}}. It has also been shown that there exist coteries that cannot be obtained from a vote assignment [2]. As will be demonstrated in this paper, non-vote assignable coteries may be needed to optimize system performance. Multi-dimensional voting is a voting-like technique that can be used to implement non-vote assignable coteries [5].

We consider a store-and-forward network with coordinators resident in some of the switching nodes. (See Figure 1.) The nodes and the links in the network are unreliable, and failure of a switching node where a coordinator resides implies that the coordinator is not accessible. Users are attached to the system by a connection to one of the switching nodes. A user becomes disconnected if the node to which he is attached fails.

The main motivation for having multiple coordinators is to enhance system availability. A system with a single

*This work was supported in part by NSF grants NCR-8604850 and CCR-8806358.

Using Checkpoints to Localize the Effects of Faults in Distributed Systems *

Mustaque Ahamad Luke Lin

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

checkpointing scheme can be used to ensure forward ress of a computation (program) even when failures oc- In a distributed system, many autonomous programs execute concurrently and obtain services from a set of d servers. In such a system, it is desirable to restrict ckpoint or rollback operation to a single program to ze the effects of failures, even when processes of dif- programs communicate with servers. This can be ved by a scheme based on message logging and consis- heckpoints when the system is deterministic. When stem (communication network or programs) is non- ministic, the semantics of the server functions should ploited to reduce the additional synchronization that to be introduced to ensure locality. We illustrate this esenting efficient algorithms for a file server that do quire the logging of messages on stable storage.

troduction

istributed program can exploit the concurrency in- in an application by executing it at many nodes. er, unlike a centralized system, it is possible that parts (nodes or communication links) of the system il while others remain operational. This could result nconsistent state in which the results of a computa- e reflected at some nodes but not at others. *Atomic ctions* [10] provide a mechanism which guarantees computation either completes at all nodes or has no n the system. The atomicity property provided by ctions masks failures from the users by undoing par- ompleted computations when failures are detected, es not promise *forward progress*. Thus, failures do ıse inconsistent executions, but they can lead to the d undoing of a computation. This can be avoided g a checkpoint and rollback scheme [14] which al- e computation to be restarted from an intermediate

entralized systems, checkpointing and rollback- y are straight forward. A process takes a checkpoint periodically by saving its state on stable storage [10]. When a failure occurs, the process rolls back to its most recent checkpoint, assumes the state saved in that checkpoint, and resumes execution. In distributed systems, checkpoints are maintained for each process in the system.

In distributed systems, there are many concurrently executing computations or programs, each of which is executed by a set of processes. Usually, there is no direct interaction between processes of different programs. However, processes of different programs may interact with a common set of server processes (e.g., file server) which implement services provided by the system. Since the execution of each program is relatively autonomous, we want to find checkpoint and rollback algorithms with the following properties, even when processes of different programs communicate with the shared servers.

- When a program wants to checkpoint its current state, processes in other programs should not be required to take a checkpoint.
- When one or more processes in a program fail, their rollback should not roll back processes in other programs.

We call these *locality* properties because they require that the checkpoint or rollback of one program does not affect others. These are necessary in large distributed systems to localize the effect of failures. Since the number of failures will increase as the system becomes larger, an uncontrolled propagation of a rollback each time a failure occurs in the system can impede forward progress. Another advantage of locality is the reduced cost of checkpoint and rollback operations because a smaller number of processes need to participate in their execution.

There are several approaches to checkpointing in distributed systems. In consistent checkpointing [1, 8, 11, 19], processes coordinate their checkpointing such that the set of checkpoints taken from all the processes forms a consistent global state of the system. A global state [4] is consistent if no message is recorded as received before it has been sent. When a failure occurs, processes roll back and restart from their most recent checkpoints. Tamir and Séquin [19]

s work was supported in part by NSF grant CCR-

Checkpointing and Rollback-Recovery in Distributed Object Based Systems *

Luke Lin Mustaque Ahamad

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

A checkpointing scheme can be used to ensure forward progress of a computation (program) even when failures occur. In this paper, we present efficient checkpoint and rollback algorithms for distributed object based systems. Previous work in distributed checkpointing has been devoted primarily to message based systems. By utilizing the structure of objects and operation invocations, efficient algorithms are developed that involve fewer nodes compared to when invocations are treated as messages and existing algorithms are used.

1. Introduction

There are two paradigms for structuring distributed systems. In *message based systems*, processes do not share memory, and communicate by exchanging messages. In *object based systems*, objects encapsulate data and define a number of operations that can be invoked by threads. A thread is an active entity that executes code in objects, traversing objects as it executes (a thread is comparable to a process, as defined in many conventional systems). In an integrated distributed system, a thread can invoke both local and remote objects in a uniform manner. Since any thread can invoke an object (when it has permission to access it), all objects logically reside in a global address space that is shared by the threads. Object based systems are becoming increasingly popular since objects provide a secure and easy to use abstraction of shared memory, which is seen by many as an attractive concept for programming distributed systems.

In an object based system, a distributed computation is executed by one or more threads. The state of an object is defined by the data encapsulated by it and threads transform the object state by possibly changing the values assigned to its data items. A thread moves from one object to the next through operation invocations, which create dependencies between different objects that may be stored at several nodes. Dependencies between computations are created when the threads executing on behalf of them invoke common objects.

In a distributed system, it is possible that some components (nodes or communication links) of the system may fail while others remain operational. This could result in an inconsistent state in which the results of a computation are reflected at some nodes but not at others. *Atomic transactions* [11] provide a mechanism which guarantees that a computation either completes at all nodes or has no effect on the system. The atomicity property provided by transactions masks failures from users by undoing partially completed computations when failures are detected, but does not promise *forward progress*. Thus, failures do not cause inconsistent executions, but they can lead to the repeated undoing of a computation. This can be avoided by using a checkpoint and rollback scheme [18] which allows a computation to be restarted from an intermediate state.

In centralized systems, checkpointing and rollback-recovery are straight forward. With this scheme, a processor takes a checkpoint periodically by saving its volatile state on stable storage [3, 11]. When a failure occurs, the processor rolls back to its most recent checkpoint, assumes the state saved in that checkpoint, and resumes execution. However, in distributed systems, asynchronous interaction among different nodes makes checkpointing and rollback-recovery more complicated. Depending on the algorithm used, a checkpoint or rollback may involve several nodes and threads. After a failure, recovered nodes must be brought back to a consistent state with nodes that did not fail.

For distributed systems, the checkpointing problem has primarily been addressed for message based systems. Although there exists a duality between message and object based systems [12, 14, 21] and it is possible to convert algorithms for message based systems so they can be used in object based systems, little or no work has been done in distributed checkpointing which exploits the structure of object based systems.

The algorithms that have been proposed can be divided into two classes: independent checkpointing, and consistent checkpointing. In the first approach, a process can

*This work was supported in part by NSF grant CCR-8806358.

CH 2877-9/90/0000/0097/$01.00 – 1990 IEEE

Implementing and Programming Causal Distributed Shared Memory*

Mustaque Ahamad *Phillip W. Hutto* *Ranjit John*

College of Computing, Ga Tech
Atlanta, Georgia 30332-0280 USA
Email: {mustaq, pwh, rjohn}@cc.gatech.edu

Abstract

Causal memory is a weakly consistent memory in which reads are required to return the value of the most recent write based on the causal ordering of read and write operations. We present a simple owner protocol for implementing a causal distributed shared memory (DSM) and argue that our implementation is more efficient than comparable coherent DSM implementations. Moreover, we show that writing programs for causal memory is no more difficult than writing programs for atomic shared memory. We believe that causal memory is an attractive target architecture for DSM systems.

1 Introduction

Distributed shared memory is an attractive abstraction because it allows processes uniform access to local and remote information. This uniformity of access simplifies programming, eliminating the need for separate mechanisms to access local state and remote state. However, consistent distributed shared memory (DSM) can be difficult to implement efficiently. Most DSM implementations to date use variants of multiprocessor cache consistency algorithms that perform poorly in high latency distributed systems. Weakly consistent memories allow implementations better suited to the high latencies encountered in distributed systems.

Traditionally, a shared memory is correct if reads return the value of the "most recent write" to the location being read. Atomic memory satisfies this "register property" by regarding reads and writes as *operation intervals* on a global time line and requiring that operations "take effect" at some point within the operation interval [17]. Under this model, each operation corresponds to a distinct point (operations may not "take effect" simultaneously) on the global time line and, for any read operation, the most recent write is well-defined. While the order of overlapping writes may not be determined until a subsequent read operation "chooses" which write is the most recent, the resulting execution must still obey the register property. Sequential consistency [12] is a weakening of atomic memory that relaxes the requirement that operations take effect during their operation intervals. Several researchers [20, 2, 6] have sought to exploit the considerable flexibility provided by sequential consistency over atomic memory yet the requirement that sequentially consistent executions appear "as if" they obey the register property is severely restrictive.

Existing implementations of consistent (atomic) DSM [15, 18] require frequent, expensive global synchronization leading to inefficiency and problems of scale. Researchers in the architecture community have also begun to question the wisdom of always maintaining strong consistency [1, 14, 9, 7]. Recent work [10] has suggested that the *principled weakening* of consistency may solve problems of latency and scale and still provide a reasonable programming model.

We explore a type of weakly consistent memory introduced in [10] that we call *causal memory*. (A formal study of causal memory is presented in [3] where the memory discussed in this paper is called *strict* causal memory.) Informally, causal memory requires that reads return values consistent with all causally related reads and writes of that same location. We say that "reads respect the order of causally related writes." Causal memory does not require all writes of a single location to be totally ordered; several processors may write a location concurrently and independently, without synchronizing. Subsequent readers may disagree on the relative ordering of these concurrent writes. Causal memory is based on Lamport's concept of *potential causality* [11]. We introduce a similar notion of causality based on reads and writes in a shared memory environment. Causal memory is also closely related to the ISIS *causal broadcast* introduced in [5]. A notion

*Funded by NSF grants CCR-8619886 and CCR-8806358.

Weakening Consistency in Distributed Shared Memories*

Phillip W. Hutto *Mustaque Ahamad*

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA
Phone: (404) 894-2593

pwh@cc.gatech.edu, mustaq@cc.gatech.edu

Abstract

We propose and advocate the use of weakly consistent memories in distributed shared memory systems to combat unacceptable network delay and to allow such systems to scale. We examine proposed memory correctness conditions and demonstrate how they are related by a weakness hierarchy. Multiversion interpretations of memory are introduced as means of systematically exploring the space of possible memories. Slow memory is one such memory that allows the effects of writes to propagate *slowly* through the system, eliminating the need for costly consistency maintenance protocols that limit concurrency. Slow memory possesses a valuable locality property and supports a reduction from traditional atomic memory. Thus slow memory is as expressive as atomic memory. We demonstrate this expressiveness by presenting two exclusion algorithms and a solution to Fischer and Michael's dictionary problem on slow memory.

*This work funded by NSF grants CCR-8619886 and CCR-8806358. Submitted to *IEEE Transactions on Parallel and Distributed Computing*. A preliminary version of this paper appears in the *Proceedings of the 10th International Conference on Distributed Computing Systems*, May 1990, Paris, France.

Shared Memory Programming in a Distributed System *

Mustaque Ahamad
Muthusamy Chelliah
Partha Dasgupta
Richard J. LeBlanc
Mark Pearson

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA
Phone: (404) 894-2593

mustaq@cc.gatech.edu

Abstract

Differing remote and local state access mechanisms in distributed computing environments make programming for concurrent execution difficult. This paper presents a novel programming paradigm that simplifies development tasks by allowing programmers to create distributed applications without having to be aware of their physical distribution, or runtime degrees of concurrency. This paradigm, used to program distributed applications for the CLOUDS distributed operating system, models these applications like centralized programs where processes communicate and synchronize using shared memory. Distribution is achieved automatically at execution time.

*This work funded by NSF grants CCR-8619886 and CCR-8806358.

Selected Publications in Full

Using Multicast Communication to Locate Resources in a LAN-Based Distributed System[†]

Mustaque Ahamad *Mostafa H. Ammar* *José M. Bernabéu-Aubán* *M. Yousef Khalidi*

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

In this paper we present a resource (e.g., file, process) location scheme which exploits the multicast communication capability of local area networks. In the scheme, the universe of resource names is partitioned into a relatively small number of groups and each group is assigned a unique address. Nodes storing the locations of resources belonging to a particular group instruct their network interfaces to receive all location messages sent to the group address. To locate a resource, a node first determines the address of the group to which the resource belongs (this can be accomplished via a well-known hash function), and a multicast message is then sent to the address. The algorithm performance is studied by means of simulation, and approximate closed form solutions are derived for systems operating at heavy and low loads. The scheme's performance is compared with that of broadcast, and it is shown that the proposed scheme performs much better than broadcast alone.

Introduction

ıe advantages offered by distributed systems include resource aring, fault-tolerance and parallel execution of a computation. ıe programming of distributed systems is more complex than ntralized ones due to the unavailability of the global state of e system. For example, in a dynamic system where resources g., files, processes) can be migrated between nodes, a user ıst program an algorithm to find the current location of a re urce needed by his or her computation. This can be avoided if ers are provided with the abstraction of a unified system where e location of resources is transparent to them. Resources are ferred to by names and, at runtime, the system determines the rrent location of a named resource.

Many schemes have been proposed for finding the location a named resource. Conceptually, there exists a database that ores the associations between resource names and their loca ıns. This database can be partitioned and stored at one or ore nodes that are called *name servers*. When a remote re urce, R, needs to be accessed, the request for its location ould be sent to a name server that stores R's location. The system must also implement algorithms to update the information stored by the name servers when resources are created or deleted or when they are migrated. To avoid this, the database can be distributed in such a way that a name server at a node maintains a list of only resources local to the node. In such a system a remote resource can be located by broadcasting its name, and having the node where the resource is located respond. This scheme is used in Clouds [DLS85] for locating remote objects. Broadcast can also be used when other schemes fail to locate a resource.

We are concerned with a distributed system that uses a broadcast bus local area network. In such an environment, all network interfaces receive every message carried on the bus. A particular message is delivered to the attached node only if it is sent to a destination address that the interface has been instructed to recognize. Such addresses will at least include the broadcast address and the node's own address. Thus, if a broadcast message is used to locate a resource, the message will be delivered to all the nodes in the distributed system. This in turn will cause all the nodes to search their local resource directories which represents a wastage of CPU time at all nodes except the one where the resource resides.

In this paper, we explore the design of a distributed name server where multicast communication is used to locate the requested resource. In such a system, a particular message sent to locate a resource will be delivered to only a subset of the nodes in the system. The availability of bus interface communications technology that supports multicast in the hardware provides the motivation for this work. Our goal is to design a location scheme that is simple from the point of view of a node that needs to find a resource but, at the same time, reduces the number of nodes that must participate in the location process.

We associate a multicast address with each resource name and this address is used to communicate with the name server of the resource. Each node receives messages sent to multicast addresses corresponding to the resources whose locations are stored by the local name server. Typically, a limited number of multicast addresses will be available at each interface for use by the resource location operations. Since the number of resources in the distributed system can be large, the resource name to multicast address mapping is many-to-one. For such a system, we present the algorithms to be executed when a resource is created, deleted or a request is made for finding its location. We also study the performance of the multicast scheme and compare it with broadcast. The cost measure used is the number of nodes that process messages sent for finding a resource or for updat-

[†]This work has been partially supported by NSF grants CCR-8806358, CR-8604850, and CCR-8619886.

ing the information stored by name servers when resources are added or deleted. We use simulation and analytical techniques to determine the cost and demonstrate that the expected cost is much smaller for the multicast scheme when it is compared with broadcast.

We do not claim that to locate resources, a distributed system should use only the scheme proposed in this paper. It can be used in conjunction with other methods, e.g., hint tables to avoid broadcasting the request when a resource is not found at its expected location.

Section 2 describes related work and the system model is presented in section 3. The algorithms that implement the multicast scheme are described in section 4. We discuss the correctness of the algorithms in section 5. Performance analysis and simulation results are described in section 6. We conclude the paper in section 7.

2 Related Work

Name servers are used for locating resources in many systems. In the Grapevine system [BLNS82], a resource name is of the type F.R where R is the name of a registry and F is the name of the resource in the R registry. Each registry has associated with it a collection of name servers. When the location of a name server for R is not known, it is found from a well-known registry wich is maintained in every name server. The Clearinghouse [OD83] system generalizes this by adding another level for naming.

If a resource is accessed at a node many times, its location can be cached so that the node does not have to consult with a remote name server each time the resource is used. Cached information is called *hints* and have been discussed in [Ter87,ABA88]. Since a resource can migrate, hints can be wrong and hence a name server should be located in that situation. In another scheme called *forwarding addresses* [Fow85], a node stores the address of the node where a resource residing at it has moved. A resource is located by following these addresses.

The broadcast scheme, where a message for finding a resource is sent to all nodes in the network, is a special case of the scheme presented in [MV85] in which a node queries a subset of the nodes to find the location of a resource. In the V system [CM86], multicast is used to communicate with the name server nodes when the resource is not found at the expected location and its name server is not known.

In the scheme presented in this paper, each node implements the scheme (it is possible to exclude certain nodes) and the set of nodes that receive a message sent to locate a particular resource depends on the resource name. Thus, the sets of nodes that process the location message for two different resources may be different. We study the relationship between the number of multicast groups supported by the hardware, the sizes of the multicast groups (the number of nodes that receive messages sent to a multicast address) and the number of resources in the system.

3 System Model

A distributed system is assumed to be a set of L (numbered from 1 to L) nodes connected by a broadcast bus. Each node contains a set of resources which can be accessed by both local and remote nodes. Each resource has a unique name which is used by the users to refer to the resource. The set of resources residing at a node is dynamic: new resources can be created and existing ones can be deleted. Resources can also be migrated between nodes.

A node consists of a processor (could also be a multiprocessor) with its own memory and a network interface that allows the processor to exchange messages with other nodes.

The network interface receives messages transmitted over the network and performs address recognition to determine if an arriving message should be delivered to the processor. The interface is also responsible for transmitting messages sent by the processor.

We assume that the network interface can recognize the unique address associated with the node, the broadcast address and a set, Φ, of multicast addresses. A message sent to the multicast address m will be delivered to a processor only if m is in its Φ. A processor can change the membership of its own set Φ. However, the number of addresses in Φ cannot be more than M. Thus, at any point in time, a node can choose to receive multicast messages sent to at most M addresses. The restriction on the size of Φ holds for currently available network interfaces.

For example, the Digital UNIBUS Network Adapter, DEUNA[1] [Deu83], supports a maximum of 10 multicast addresses (there exist interfaces that support a larger number of addresses but the set of addresses allocated to the location subsystem will be limited in size). We also assume that if a node sends a message which generates a response, the sender will receive the response in at most δ seconds. This allows the use of timeouts for deciding when not to wait for any more responses.

We assume that the operating system at each node, in addition to other functions, implements a *resource management subsystem*, RMS, and a *location subsystem*, LS. RMS handles the creation, deletion and migration of resources and stores information about all resources that are currently resident at its node. When a user needs to access a remote resource, RMS communicates with LS, which finds the current location of the remote resource. We assume that identical copies of RMS and LS execute at each node and RMS informs its local LS when a resource is created or deleted.

4 Location Subsystem

The LS executing at a node communicates with its local RMS and the LS at other nodes to implement a distributed name server. In this section, we describe the data structures maintained by each LS and its interface with the RMS. We also describe the algorithms executed when a resource is created,

[1] DEUNA and UNIBUS are trademarks of Digital Equipment Corporation.

deleted or its location needs to be found. Since some of these functions can be executed concurrently at different nodes, their code must use some synchronization mechanism to assure atomicity when it is required. We do not include the code for synchronization to avoid the unnecessary complexity. Also, it is assumed that no node failures occur. The effect of failures on the scheme and their handling is described in a [AABK87].

4.1 Location Subsystem Data Structures

Each LS maintains a directory of $<$ *resource name*, *node* $>$ pairs and a multicast table. A multicast table entry consists of a multicast address and a count. A mapping (e.g., a hashing function), ω, which is well known, is used by LS to map a resource name to a multicast address. For each R such that $< R, i >$ is in the directory, there must be an entry in the multicast table with $\omega(R)$ as the address. The count field of this entry is the number of resources in the local directory that map to the address $\omega(R)$. Initially, the address field in each entry of the table is set to a multicast address E which is not in the range of the mapping ω. As will be seen later, the resources whose names are stored in the directory can be both local as well as remote. Each address in the multicast table at a node is also added to its Φ and hence the size of the multicast table cannot exceed M. A multicast message sent by LS with m as its address will be received by a node if there is a resource, R, in its directory such that $\omega(R) = m$ (it is assumed that m belongs to a set of multicast addresses that are used only by the *location subsystem*).

4.2 Location Subsystem Calls

The RMS at a node not only calls the local LS for locating a remote resource, it also makes a call to LS when a resource is created or deleted. The functions implemented by LS that are called by RMS are defined below. To implement these functions, LS may have to communicate with its peers at other nodes. A message sent by LS contains its destination and source addresses, a type and data (if any) depending on the type of the message. The types of the messages used by LS and the data contained in them is described in the code for the functions. We assume that these functions are called at node i ($1 \leq i \leq L$).

- `AddResource`(R : ResourceName)

  This function is called by RMS when the resource R is created. This makes R available to remote nodes that can locate it by requesting their LS. Since the multicast table size is limited, the resource name and its location may have to be added to the directory at some other node. When it is not possible to add $< R, i >$ to the local directory, the `CreateSpace` function is called, and returns the node that can add $< R, i >$ to its directory and $\omega(R)$ to its multicast table. We describe the `CreateSpace` function later.

```
function AddResource (R : ResourceName)
  begin
    if ω(R) is in the multicast table then
      increment the count of the entry having
      address ω(R); add < R, i > to directory;
    else if entry in the table with address E then
      change the address in the entry from E to ω(R)
      and make its count 1;
      add < R, i > to directory;
    else
      j := CreateSpace(R);
      if i ≠ j then
        send a message of type AddReq to node j ,
        with < R, i > as data;
      else
  (* A free entry is created in local multicast table *)
        change the address in the entry from E
        to ω(R) and make count 1;
        add < R, i > to directory;
  end;
```

- `FindResource`(R : ResourceName)

  This function is called by RMS to find the location of the remote resource R. If R exists at some node currently, the address of that node is returned by this function.

```
function FindResource (R : ResourceName)
  begin
    if < R, j > in the directory then
      return(j);
    else
      send a FindReq message to address ω(R) with R
      as data; wait for FindResp message;
      j := address of node sending FindResp message;
      return(j) ;
  end;
```

- `DeleteResource`(R: ResourceName)

  This function is called when RMS needs to delete R. RMS asks the local LS to find its location node, j, and delete the resource name and its multicast address at the node where they are stored. We do not consider the messages sent by RMS to its peer at node j to actually delete the resource.

```
function DeleteResource(R : ResourceName);
  begin
    if < R, j > in directory then
      delete < R, j > from directory;
      decrement the count in the entry with address
      ω(R) in the multicast table;
      and when count becomes 0, change ω(R) to E;
      return(j)
    else
      send a DeleteReq message to ω(R)
      with R in the data field; wait for a
      DeleteResp message returning node name j;
      return(j)
  end;
```

LS does not provide a function to be invoked when a resource is migrated. RMS can inform LS of the migration by deleting the resource at its current node and adding it at the new node by calling the functions described above.

4.3 Internal Functions of The Location Subsystem

We now describe the `CreateSpace` and the `MessageHandler` functions. These functions are internal because no other component of the system has access to them. Again, we assume that the functions are executed at node i.

- `CreateSpace`(R)

  The `CreateSpace` function is called by LS at node i when it cannot add $< R, i >$ to its directory because all addresses in the entries of the multicast table are different from $\omega(R)$ and E. Since *FindReq* messages for R are addressed to $\omega(R)$, the node where the location of R is stored must have $\omega(R)$ in its multicast table. The `CreateSpace` function finds a node where either $\omega(R)$ is in the multicast table or there is an entry with address equal to E. When this cannot be done, it creates an entry with address E at some node by moving resource names from the node's directory to some other node. The range of ω has to be restricted to assure that `CreateSpace` returns a node with this property. We discuss this in a later section.

```
function CreateSpace(R)
  begin
(* Check if some node has ω(R) in multicast table *)
    send a SpaceReq message to ω(R);
    wait for SpaceResp message to arrive for δ time;
    if one or more SpaceResp messages arrive then
      choose one and let j be the sender
      of the chosen message;
      return(j);
(* Check if some node has address E
in its multicast table *)
    send a SpaceReq message to E;
    wait for SpaceResp messages to arrive for δ time;
    if one or more SpaceResp messages arrive then
      choose one and let j be the sender
      of the chosen message;
      return(j);
(* No node has ω(R) in its table and all tables are full *)
    send a TableReq message to the broadcast address;
    wait for TableResp messages to arrive for δ time;
    let j and k be two nodes such that multicast tables
    received from them in the TableResp messages
    have a common multicast address[1] m.;
    send a MoveDirEntryReq message to j with k and m
    in the data field;
    wait for a MoveDirEntryResp message;
    return(j) ;
```

- `MessageHandler`(msg)

  The `MessageHandler` function is executed by LS at node i when a request message arrives for LS. This message may have been sent to the unicast address of i or a multicast or the broadcast address. Since only request messages arrive asynchronously, we show the handling of these messages. The response messages are received when LS sends a request message and their handling is described in the code of the functions.

```
function MessageHandler(msg : Message)
begin
  j := sender of msg;
  case msg.type of
    AddReq:
      < R,k > := data received in msg;
      if ω(R) is address in an entry in the table then
        increment the count in the entry
        with address ω(R); add < R,k > to directory;
      else if there is an entry with address E then
        change the address in the entry
        from E to ω(R); make its count 1;
        add < R,k > to the directory;
    FindReq:
      R := resource name received in msg;
      if < R,k > in the directory then
        send k in a FindResp message to j;
    DeleteReq:
      R := resource name received in msg;
      if < R,k > in directory then
        delete < R,k > from directory;
        decrement the count in the multicast
        table entry having address ω(R);
        if count becomes 0 then change ω(R) to E;
        send k in a DeleteResp message to j;
    SpaceReq:
      send a SpaceResp message to j;
    TableReq:
      send multicast table in TableResp message to j;
    MoveDirEntryReq:
      m := multicast address received in msg;
      k := node address received in msg;
      for each < R,l > in the directory
      such that ω(R) = m do
        send an AddReq message to k with < R,l >
      as data; change ω(R) to E in the multicast table;
      send MoveDirEntryResp message to j;
```

2 Correctness

The correctness requirement for the multicast based scheme is that when a resource exists (it has been added by calling the function `AddResource`(R) and it has not been deleted by calling `DeleteResource`(R)), then executing `FindResource`(R) at any node must return the current location of R. Let i be the node where `FindResource` is executed and let j be the current location of R. If $< R, j >$ is not in the directory at node i then a *FindReq* message is sent to $\omega(R)$. Thus, the location of R will be returned by the `FindResource` call if the node where $< R, j >$ is stored in the directory has $\omega(R)$ in its multicast table. This will guarantee that the *FindReq* message for R is received by the node that stores its location. Since $< R, j >$ is added to the directory at a node only when either $\omega(R)$ is in the multicast table or there is an entry with address E which is changed to $\omega(R)$ (`AddResource` function and handling of *AddReq* message in `MessageHandler`), the correctness follows if we can demonstrate that $< R, j >$ is added to the directory at some node as

[1] An alternative way of getting two nodes which share a multicast address is to poll nodes one at a time until two nodes with a common address are identified.

result of executing AddResource.

If $\omega(R)$ or E is the address in an entry of the multicast table t node i when AddResource is executed, $< R, i >$ is added to ie local directory. Otherwise, $< R, i >$ is sent to node j which returned by the CreateSpace function. If j is the address of ie chosen node that responded to the *SpaceReq* message sent) either $\omega(R)$ or E then $< R, i >$ is added to the directory at ie responding node. When no nodes respond to the *SpaceReq* essages sent to these addresses, then multicast tables at all odes are full (there is no entry with E as the address) and one of the tables has an entry with the address $\omega(R)$. In this se, all multicast tables are collected at node i and two tables aving a common multicast address are found. To guarantee iat there exist two such tables, we need to restrict, K, the nge of ω. Since the multicast table size is M and there are L odes, if $K \leq L \cdot M$ then two tables will have a common address hen all tables are full and none of them has the address $\omega(R)$. his follows because otherwise $K \geq L \cdot M + 1$ (all addresses in e multicast tables are distinct and different from $\omega(R)$) which a contradiction.

Once two nodes such that their multicast tables have a common address, a, are found, the entry containing a at one node freed by sending all resource names mapping to the address a the other node that sent the table with address a in it. The source name entries deleted from the directory of one node e added at the other node because the multicast address corsponding to the resource names is in the table at the other de. R is added to the directory at the node where the free en- is created. Thus, when AddResource(R) is called, the name

id location of R are added to the directory of some node which is $\omega(R)$ in its multicast table.

Performance Study

o study the performance of the location scheme presented above e will use a simulation model of a system that uses the multist location algorithm. The simulation results will provide us ith an understanding of how the performance of the proposed heme is affected by various parameters and how it compares ith the use of broadcast to locate a resource. We will also esent analytic results for light and heavy load approximations.

1 Simulation Model

or the purpose of the simulation we make the assumption that id, *delete* and *add* operations occur independently of each other. equests to *find* and *delete* a particular resource are only allowed hen the resource has been added but not yet deleted. Resources e added to the system as a Poisson process with rate γ. An *id* request is equally likely to arrive at any of the nodes. A source being added is equally likely to have its name mapped any address (this is a property of the function ω). Thus the tal arrival rate of *add* requests per node per address is $\frac{\gamma}{LK}$ ecall that L = number of nodes, K = number of multicast ldresses in the range of ω, and M = size of multicast tables). nce a resource is added, it will reside in a node for a time that exponentially distributed with rate μ. A *delete* request for a particular resource is equally likely to occur at any node in the system. Once a resource has been added, *find* requests are generated for it at a rate λ until it is deleted. The interarrival time of find requests for a particular resource is exponentially distributed and a *find* request is equally likely to arrive at any node in the system.

We are interested in studying the system in the steady state, and in that state, the rate of resources leaving the system will be γ, and the average time spent by a resource in the system is given by $\frac{1}{\mu}$. Thus, the average number of resources in the system, $\bar{r}$, can easily be computed by applying Little's Law[Lit61].

$$\bar{r} = \frac{\gamma}{\mu} \tag{1}$$

The simulation closely follows the steps of the algorithms presented in section 4. In the definition of the CreateSpace function, three phases can be distinguished. In the first phase, a *SpaceReq* message is sent to a multicast address (different from E) and one of the nodes responding to it is selected. In the simulation, it is equally likely that any particular node be selected from the set of nodes having the multicast address in their tables. In the second phase, one of the nodes with empty multicast table entries has to be selected. Again, any node is equally likely to be selected. Finally in the third phase both a multicast address and two nodes belonging to its multicast group have to be selected. It is equally likely that any particular multicast address will be selected out of those which are in tables at more than one node. Any pair of nodes with that address is also equally likely to be chosen.

We assume in the discussion that the system will be fault-free. In particular it will always be possible to add a new resource, and resources for which *find*'s and/or *delete*'s arrive, must have already been added.

6.1.1 Cost Calculation

We describe the performance of the multicast scheme in terms of the cost of certain operations. This cost is defined as the number of messages delivered and processed by nodes in the system. Thus, for example, if during an operation a message is sent to a multicast group consisting of 3 nodes, and in turn one of the nodes sends back an acknowledgement message, the cost of this operation would be 4 under the proposed metric. This metric properly reflects the amount of total CPU time used by the location system. In calculating our costs we are concerned only with the function that terminates by returning the location of the resource to the RMS. Additions and deletions of resource references are considered, however the costs of addition, deletion or usage of the resource itself are not.

A detailed description of how the costs are computed for each of the *location subsystem* operations is included in [AABK87]. The cost of an operation depends on the size of the multicast group to which a message is sent to implement the operation. The multicast group size depends the number of nodes in the system, L and the multicast table size M.

With the same assumptions about system behavior as we made above, we can find the average cost for each operation type (and the total combined average for all types) when broadcast is used as the only location method. The probability that a

resource, chosen at random, be local will be given by $\frac{1}{L}$, thus the average cost of a *find* (or *delete*) operation will be given by,

$$C_f = C_d = L \cdot \frac{L-1}{L} = L - 1 \quad (2)$$

The cost of additions will be always zero, because the resource reference is stored only locally. As shown in [AABK87], the combined cost for the three types of operations will be,

$$C_B = (L-1)\frac{\lambda + \mu}{\lambda + 2 \cdot \mu} \quad (3)$$

where λ , μ and L are as defined earlier.

6.2 Simulation Results

We performed two sets of simulations. In the first set, the simulations were run for a system consisting of 20 nodes in which each node's multicast table could hold up to 10 multicast addresses. In the second set, the value of K was fixed to 100, and the value of M varied from 5 (its minimum) to 100.

For the first set, different values for K have been considered, covering all its possible range (notice that it is necessary that $K \leq L \cdot M$). The maximum load (average number of resources in the system) considered was 2000 (that is 100 resources per node on the average when the multicast table size, M, is 10). Due to the fact that the cost of *delete* operations varies similarly to the cost of *find*'s, we have only shown the latter's costs.

In figure 1, we show the variation of the average cost of *find* operations. We plot the variations for several values of K. It can be observed that the costs reach a definite asymptotic value at high loads, and this value is reached relatively fast as the load is increased. It can also be seen that for large K (close to the maximum), there is a relative maximum in the cost curve (although it is not very pronounced). To understand this behavior, we have to consider what happens when the load varies from 1 to 2000. We start by pointing out that with our cost measure, the average cost of a *find* operation will increase with the number of nodes receiving messages sent to a given multicast address. The larger the number of multicast addresses, the smaller the number of nodes with a given address. At load 1 there is, on the average, a number of nodes close to one which contain a given multicast address. As the number of resources in the system increases, the number of nodes containing resources that map to a given multicast address will increase while there is enough room in the tables to store the multicast addresses of all existing resources. Thus the cost of *find* requests will also increase. As the multicast tables start getting full and $K > M$, the multicast addresses will compete with each other for a place in the tables as a result of calls to the `CreateSpace` function. This will in general decrease the number of nodes in a given multicast group: references of resources that map to a particular address will be moved by using the *MoveDirectory* message and will be collected at a small number of nodes, thus decreasing the cost of a *find*. When $K \leq M$ the cost curves are monotonically increasing. This is because there is never competition between the multicast addresses, and, in the limit, all addresses are in the multicast tables of all the nodes, thus making any multicast message equal in cost to a broadcast.

As shown in figure 1, the larger the number of addresses, K, the lower the cost of *find*. This is a direct consequence of the fact that increasing the number of multicast addresses reduces the number of multicast table entries available per address, thus reducing the number of nodes in a particular multicast group. For the system being considered, the average cost of a *find* operation when only broadcast is used to locate objects is given by equation (2) and equal to 19. It can be seen that even for relatively low values of K ($K = 20$), the cost of using the multicast scheme is slightly more than half that of broadcast for heavy loads (it is even lower for low loads). When K is incremented to 50, the cost reduces to approximately one fifth of the broadcast cost. Thus the multicast method compares very favorably with respect to broadcast for *find* operations (the same can be said about *delete* operations).

Figure 2 plots the average cost of *add* operations versus the average number of resources in the system. For $K \leq M$ this cost will be zero (there is always room in the multicast table to store the address of a new resource). For any given K, the cost seems to vary similarly to the cost of *find*'s. The biggest difference consists of the fact that at low loads, the larger K, the larger the cost, whereas at high loads the opposite is true. This happens because at loads high enough so that addresses have already started to compete for multicast table entries, but low enough that the number of nodes in each multicast group has not yet been balanced, the likelihood of a totally new address coming in the system is high, thus forcing the execution of the `CreateSpace` function up to its second phase. Once the number of nodes per multicast address starts balancing, however, all multicast addresses will have at least one entry in the multicast tables, and the larger the K, the lesser the number of nodes containing any particular address. Thus, the cost of executing the `CreateSpace` protocol decreases with K, and, according to the figures, although the probability that the `CreateSpace` protocol be executed increases with K, its cost becomes low enough as to make it cheaper for higher values of K. For *add* operations, the multicast scheme clearly performs worse than broadcast, whose cost is zero.

We call the ratio of *add* request rate to *find* request rate the *operation mix*. The actual operation mix will not affect the costs of *find*, *delete* and *add* operations at any given load, however it will affect the overall average cost for all operation types. In figure 3 we show the variation of the overall average cost for all operation types for a system in which the operation mix is 1 : 40. It can be seen that the variation of the costs follows closely the one observed in figure 1, which is due to the fact that *find* operations are the ones contributing most to the overall cost. The average overall cost for broadcast (as derived in the last subsection) would be slightly less than 19, and the overall cost of the multicast scheme still is only slightly higher that half the overall cost of the broadcast scheme for $K = 20$, and much lower for higher values of K.

In figure 4 we plot the variation of the average cost of *find* operations against the number of multicast addresses, K, for some values of the load. It can be seen that the cost falls sharply as K increases. It can also be observed that for K close to its maximum (200) higher loads lead to somewhat lower costs of

find.

In figure 5 we plot the variation of the cost of *add* operations against the number of multicast addresses, K. In this figure, the effects seen in figure 2 are made more apparent: at medium loads, the larger K the larger the cost of *add* operations.

In figure 6 the variation of the average cost of *find* operations is plotted as M, the size of a node's multicast table, increases. The value of K for all curves is set to 100. At low loads, the cost does not seem to depend on the value of M (this is in agreement with the results obtained for the low load approximation, see next section). In general, for all loads, the cost will increase until a certain value for M is reached. Thus, by increasing M sufficiently, the system can be made to work in the "low load range".

A similar effect can be observed in figure 7, where the cost of *add* operations is plotted against M. Here, again, we observe that for sufficiently large M the system starts operating in the "low load range" (characterized by the cost of *add* being close to zero). This value for M coincides with the one observed on the plot for the cost of *find*.

6.3 Approximate Analysis

The results of the simulation indicate that the system seems to be operating mainly in two modes: at low loads, the cost increases rapidly, whereas after a certain value of the load its behavior changes radically and the system stabilizes with an almost constant cost. This suggests a description of the system's behavior at heavy and low loads will be useful to understand the system's overall behavior.

It is possible to provide models which will approximate the behavior of the algorithms for low loads. Such analysis will provide us with closed form expressions for the costs. It is also possible to obtain models which provide upper and lower bounds on the costs when the system is operating at heavy loads.

In [AABK87], we derive the expressions for the approximate costs at low loads. This is achieved by assuming that there is always room in the multicast tables to store the address to which a resource maps and, thus, all resource references are stored at the node where the resource resides. We obtain the following results for the average costs (note: C_a, C_d and C_f stand for the average costs of *add*, *delete* and *find* operations, respectively. C represents the overall average cost).

$$\begin{aligned} C_a &= 0 \\ C_f &= (2 + (L-2)(1 - e^{-\frac{r}{K\tau}}))\frac{L-1}{L} \\ C_d &= C_f \\ C &= \frac{\mu + \lambda}{\lambda + 2 \cdot \mu} C_f \end{aligned} \qquad (4)$$

Notice that the costs do not depend on the value of M. Also note that as $\bar{r} \to \infty$, the costs in (4) approach the respective costs for the broadcast approach.

In [AABK87] we also derive upper and lower bounds for the different costs in the heavy load limit. Under this limit the system is assumed to have reached a given configuration for its multicast tables. In this configuration, all multicast addresses are stored in at least one entry of some multicast table. The configuration, once reached, does not change unless the load decreases. For *add* and *find* operations we derived upper and lower bounds for the limit of the cost at heavy loads. Denoting the upper and lower bounds of an operation o by C_o^u and C_o^l respectively, we have,

$$\begin{aligned} C_a^u &= (2L-1)\frac{M}{K} + 1 \\ &\quad - \frac{2}{LK}\left[LM + \lfloor\frac{LM}{K}\rfloor\left(2LM - K(\lfloor\frac{LM}{K}\rfloor + 1)\right)\right] \\ C_a^l &= (2L-1)\frac{M}{K} + 1 - \frac{2}{LK}(qL^2 + (p+1)^2 + K - q - 1) \\ C_f^u &= (L-1)\frac{K + M(L-2)}{LK} - \frac{M}{LK} \\ &\quad + \frac{1}{L^2K}(qL^2 + (p+1)^2 + K - q - 1) \\ C_f^l &= (L-1)\frac{K + M(L-2)}{LK} - \frac{M}{LK} \\ &\quad + \frac{1}{L^2K}\left[LM + \lfloor\frac{LM}{K}\rfloor\left(2LM - K(\lfloor\frac{LM}{K}\rfloor + 1)\right)\right] \end{aligned}$$

Where $q = \lfloor\frac{LM-K}{L-1}\rfloor$ and $p = LM - K - q(L-1)$. For *delete* operations we were able to obtain the heavy load limit given by,

$$C_d = (L-2)\frac{M}{K} + 1$$

In figure 8 we plot the low load limit value and the heavy load bounds for the cost of *find* for $K = 20$. We can see that the low load limit fits the simulation curve at low loads. As the load increases, the simulation curve eventually enters the zone between the upper and lower bound approximations. A close match with the simulation results has also been observed for the cost of the other operations and for all the values of K we have studied.

7 Concluding Remarks

In this paper we presented the algorithms necessary to implement a simple location scheme based on multicast communication. To analyze its performance, a simulation model was developed which closely followed the steps of the algorithms. The simulation results showed that the scheme had a lower cost than broadcast alone. In order to predict the costs of the scheme for cases not included in the simulation, analytic results were obtained which approximate the behavior of the system at low loads, and provide tight upper and lower bounds on the costs incurred when using this location scheme on systems operating with a large number of resources.

In all cases considered, the cost of *find* operations using the multicast scheme is lower than if broadcast were used instead. Even when the number of multicast addresses is less than or equal to the number of entries in the multicast table, the multicast scheme presented in this paper has a lower cost than broadcast for low values of the load. Even though the cost of *add* will always be worse for the multicast scheme (for broadcast its cost will always be zero), the overall cost still favors the multicast scheme for large enough values of K. Since the value of K is

constrained by M, the larger the size of the multicast table supported by the bus interface, the larger K can be made.

References

[AABK87] M. Ahamad, M.H. Ammar, J. Bernabéu-Aubán, and M.Y. Khaldi. *A Multicast Scheme for Locating Objects in a Distributed Operating System.* Technical Report GIT-ICS-87/01, Georgia Institute of Technology, January 1987.

[ABA88] M. Ammar, J. Bernabéu-Aubán, and M. Ahamad. Using Hint Tables to Locate Resources in Distributed Systems. In *INFOCOM'88*, IEEE, March 1988.

[BLNS82] A. D. Birrel, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[CM86] D. R. Cheriton and T. P. Mann. *A Decentralized Naming Facility.* Technical Report STAN-CS-1098, Department of Computer Science, Stanford University, February 1986. Revised version to appear in ACM Transactions on Computer Systems.

[Deu83] *Deuna Users Manual.* Digital Equipment Corporation, 1983.

[DLS85] P. Dasgupta, R. J. LeBlanc, and E. Spafford. *The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System.* Technical Report GIT-ICS-85/29, Georgia Institute of Technology, 1985.

[Fow85] R.J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD Thesis 85-12-1, University of Washington, December 1985.

[Lit61] J.D.C. Little. A Proof of the Queueing Formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[MV85] S.J. Mullender and P.M.B. Vitányi. Distributed match-making for processes in computer networks. In *Fourth ACM Symposium on the Principles of Distributed Computing*, ACM, Minacki, Ontario, August 1985.

[OD83] D.C. Oppen and Y.K. Dalal. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3):230–253, July 1983.

[Ter87] D.B. Terry. Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):48–54, January 1987.

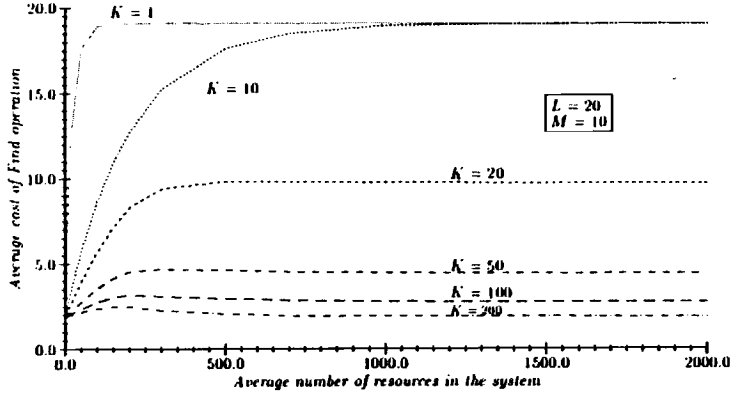


Figure 1: Average cost of *find* versus average number of resources

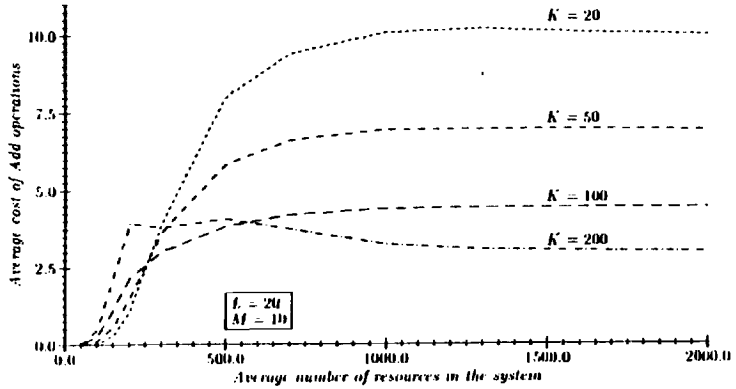


Figure 2: Average cost of *add* versus average number of resources

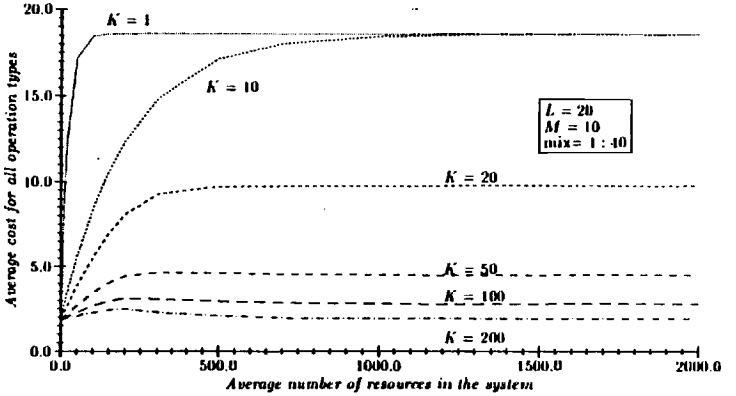


Figure 3: Average overall cost versus average number of resources

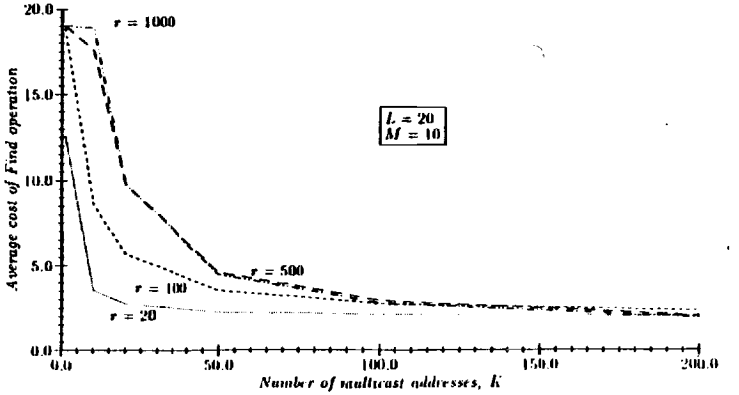


Figure 4: Average cost of *find* versus number of addresses

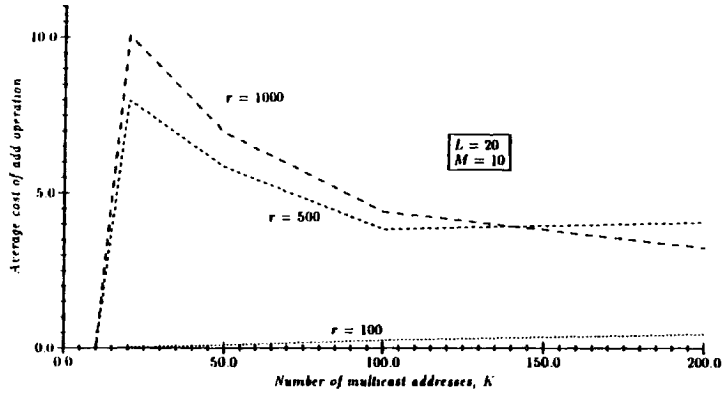


Figure 5: Average cost of *add* versus number of addresses

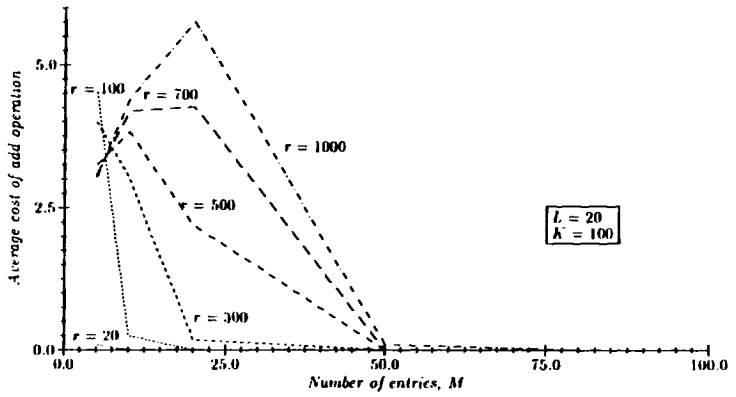


Figure 7: Average cost of *add* versus number of entries

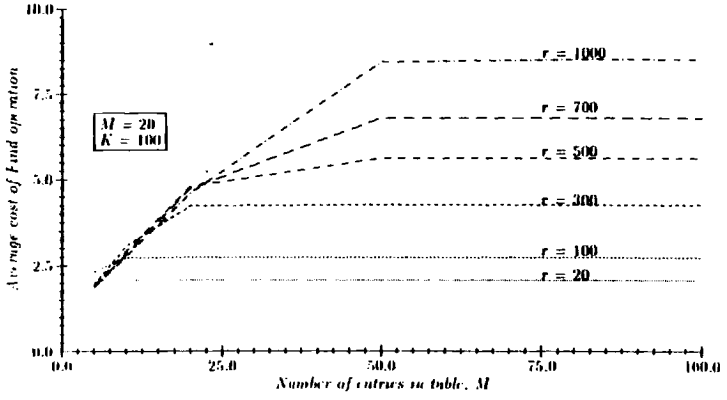


Figure 6: Average cost of *find* versus number of entries

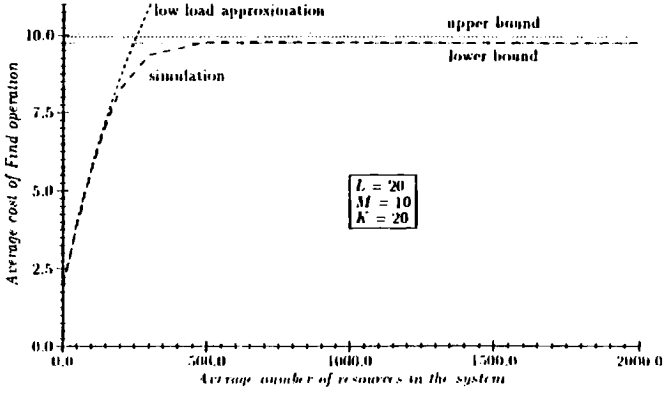


Figure 8: Comparison of analysis with simulation

To appear in Acta Informatica

Resource Finding in Store–and–Forward Networks*

José M. Bernabéu–Aubán
Mustaque Ahamad
Mostafa H. Ammar

Abstract

We present a model of searching for a resource in a distributed system whose nodes are connected through a store–and–forward network. Based on this model, we show a lower bound on the number of messages needed to find a resource when nothing is known about the nodes that have the current location of the resource. The model also helps us to establish results about the time complexity of determining a message optimal resource finding algorithm when the probability distribution for the location of the resource in the network is known. We show that the optimization problem is NP-hard for general networks. Finally we show that optimal resource finding algorithms can be determined in polynomial time for a class of tree networks and bidirectional rings. The polynomial algorithms can be used as a basis of heuristic algorithms for general networks.

*This work was supported in part by NSF grants CCR–8806358 and NCR–8604850.

1 Introduction

Distributed systems need to implement algorithms for finding the location of remote resources to reduce the complexity of their use. We investigate the communication cost of resource finding algorithms in a store–and–forward network. We consider two situations. First, we investigate the message cost of resource finding when a searcher node does not know where information about the current location of the resource resides. Such a situation can arise in a distributed system with the commonly used schemes such as name servers [1] or hint tables [2]. For example, when the name server node fails, the algorithm used by the searcher node (the node that wants to find the resource location) must work without knowledge about the nodes that are likely to know the resource location.

It may be possible to reduce the communication cost of resource finding if statistical information (e.g., a probability distribution) is maintained about the nodes that are likely to know the location of a resource. Such information can be derived from usage and migration patterns of resources in the network. For example, when the resource is an idle node, nodes having high job arrival rates will be less likely to be available when the resource is needed. When the statistical information is available to a searcher node, it should attempt to query the nodes in an order that minimizes the communication cost. We investigate the time complexity of algorithms that can be used to determine such an ordering of the nodes.

In the resource finding algorithms we consider, a searcher follows the links of the network from one node to another until a reference (identity of the node where the resource resides) to the requested resource is found. Thus, we only consider *serial searches* in which only one agent is active at any given time. To reduce the time needed to find the location of a resource, it would be desirable to start several searchers at the same time exploring different sets of nodes. Multiple agent searches have the added complexity of synchronization between the searchers, e.g., if one searcher finds a reference then others need to be informed in order to halt their searches. The results obtained here for serial searches will inevitably form the basis of an understanding of multiple agent searches. Furthermore, we will see that even with this simple model of resource finding, the problem of finding the order in which the nodes should be searched to minimize message cost is intractable in most networks. We conjecture that a more complex search model that includes multiple agents and coordination will render the problem even harder.

The following summarizes the results we have obtained for serial searches:

- When the searcher has no information about the location of a resource (or of its references):
  - The average number of messages used in searching for the resource is $\Omega(\sqrt{\frac{\mu}{\lambda}N})$, where N is the number of nodes in the network, μ is the rate at which the resource moves and λ is the rate at which requests for the resource arrive. This lower bound includes the messages needed to update the references to the resource.

- For arbitrary networks, the average number of messages needed to find the reference for a resource having n references in the network has an upper bound of $N - n$ (this bound is tight).

- When the searcher has a probability distribution describing the likelihood of a particular node knowing the location of a resource:
  - The problem of determining the optimal way of searching an arbitrary network to minimize the average number of messages used is NP-hard, even when there is only one reference for the resource. The problem remains NP-hard, even when the network is completely connected, when the cost of sending a message through a network link is not the same for all links.
  - It is shown that when the probability distribution is uniform, the optimal way to search a tree network in which the cost of traversing any link is the same, is by following a depth–first traversal.
  - An algorithm is developed for general tree networks which improves on exhaustive search. Such an algorithm has exponential time complexity for general trees, although its complexity becomes polynomial for special classes of trees. These polynomial algorithms can form the basis of efficient heuristic search techniques in arbitrary networks. Also, as a special case of the application of the algorithm developed for trees, we show a polynomial algorithm to find the optimal way to search for a resource in a bidirectional ring network.

The resource finding problem has been addressed by several researchers. In [3] and [5], distributed methods suitable for store–and–forward networks are presented and it is shown that the average cost of the methods considered is $\Omega(\sqrt{N})$ when the ratio of the request rate and the movement rate is a constant ($\mu/\lambda = c$ where c is a constant). The forwarding addresses based protocols studied in [5] require that all nodes store a node address for each of the resources and hence they could have a high storage cost. In contrast to serial search considered here which is sequential, in the method in [3], a node searches for a resource by simultaneously sending a request message to a predetermined set of nodes. The message complexity is $\Omega(\sqrt{N})$ for both of these methods. This cost could be reduced by keeping information about the nodes that are likely to know the location of the resource. The problem investigated in this paper, of determining the optimal way to search the network when such information is available for the resource's location at each node, has not been addressed in the literature.

The problem of searching has also been addressed in the literature in a different context [6, 7, 8, 9, 10, 11, 12]. However, the solutions obtained are not applicable to resource finding in distributed systems (e.g., in [11] the problem of determining the smallest number of searching agents to capture an evading target in a graph is studied). There exist other schemes which are useful in a particular type of network (e.g, a local area network) [13, 14]. These schemes may not be efficient if used in a store–and–forward network.

In section 2 we introduce our model of the system and a precise characterization of *serial search*. In section 3 we analyze the average cost of serial search when the searcher has no

knowledge about the location of the resource. In section 4 we study the problem of finding an optimal way to traverse the network when the searcher has the probability distribution of the resource being at a certain node in the network. The paper is concluded in section 5.

2 System Model

We model the system as a collection of N nodes connected by a store–and–forward communications network. Thus we can represent the system by a graph $G(V, E)$ in which the vertices represent the nodes and the edges represent the communication links in the network. A node's CPU is responsible for both switching arriving messages to appropriate outgoing links and for resource table look up, if a message requesting the location of a resource is received. With each edge e we will associate a cost $c(e)$, which is incurred every time a message traverses the edge. This cost is intended to represent the bandwidth cost associated with traversing the corresponding link. A distributed system will be represented by $G(V, E, c)$.

Each resource in the system resides in one of the nodes, and for each resource, n different nodes store a reference to the resource's current location. The node where the resource resides contains one of the references. The set of nodes containing references to a given resource is called the *well–informed set* of the resource (this set is similar to the set of nodes where the location of a resource is posted in [3]). No other node outside of the well–informed set of a resource has any knowledge about where the resource or any of its references are. We will assume that the references are instantaneously updated when a resource moves, which happens at rate μ. We will further assume that the locations of the resources are requested with rate λ. For each $A \subset V$ such that $|A| = n$, we will denote by $Q(A)$ the probability that A is the well–informed set of the resource when its location is requested at a node not in A. This distribution may not be known to searcher nodes.

2.1 Search Model

We can visualize the serial search process as one in which a searcher agent starts at a node and not finding a reference to the resource, decides on the next node to be visited. It then traverses a path leading to the chosen node in order to search there for the resource reference. This process is repeated until a node storing a reference to the resource is found, at which point the search terminates. When the objective is to find a reference to the resource incurring the least communication cost, the procedure used by the searcher to make a decision on the next node to consult has the following natural properties.

1. Once a node has been consulted, it should not be consulted again.
2. Once the searcher decides to search a new node, it will go to it through a shortest path, and all nodes along the path will have been already searched (otherwise the next node to search would be one of them).

3. As a consequence of the above, no shortest path from the current node to the next node selected can contain a not–yet–consulted node.

Serial search can be seen as a rule which selects the next node to be consulted based on the past history of the search. Since the only information the searcher can get by consulting a node not in the well–informed set is that the node does not have a resource reference, the sequence in which the nodes should be consulted can be laid out statically from the beginning. Therefore, we can formalize serial search using the notion of a *walk* in a graph that defines the order in which the searcher will visit the nodes until a reference to the resource is found. If the searcher is at node v_i and it next wants to visit node v_j, the walk would include the nodes on the path from v_i to v_j when $\{v_i, v_j\} \notin E$. When the search plan corresponds to walk $(v_0, v_1, ..., v_\ell)$ ($\{v_i, v_{i+1}\} \in E$ for $i < \ell$), v_i needs to search for a reference only if it has not been visited previously in the walk. We call v_i a *first visit* if $v_p \neq v_i$ for all $p < i$. On all visits to a node except its first visit, a searcher is merely "switched" to the next node along the walk. A search terminates at a first visited node where a reference to the resource is found. For a walk, w, we define $V(w)$ as the set of all nodes that can potentially be visited by the walk.

Although a search plan corresponds to a walk in a graph, we want to consider only those walks which correspond to search plans that satisfy the properties described above. For example, if $(v_0, v_1, ..., v_\ell)$ is a walk such that v_ℓ is not a first visit, there is no need to traverse the link $\{v_{\ell-1}, v_\ell\}$ since the resource was not found at node v_ℓ previously. In fact, only a subset of the possible walks in a graph correspond to search plans and we will call such walks *serial traversals*. The properties that are satisfied by serial traversals are stated in the following definition.

Definition 1 *A serial traversal in $G(V, E, c)$ is a walk $s = (v_0, v_1, \ldots, v_l)$, that satisfies the following conditions,*

(a) For all i, j, s.t. $i < j$, if none of the v_m, for $i < m < j$ is first visited, the walk $(v_i, v_i + 1, \ldots, v_j)$ is a shortest path between v_i and v_j.

(b) v_l is a first visit.

We call a serial traversal *s complete* if $V(s) = V$. Thus, a searcher can potentially visit all nodes if the serial traversal it is using is complete. If the locations of the resource references do not change during the time the search is conducted, a reference will be guaranteed to be found when a complete serial traversal is used (we address migration later). The set of complete serial traversals of a graph starting at node $v \in V$ will be denoted by $\mathcal{C}(G, v)$.

It is possible that if Q is known, it may not be necessary to visit some nodes because they do not store a reference to the resource. For example, if $|A| = 1$, $Q(A)$ becomes the probability of a particular node storing a reference to the resource. If $Q(A)$ is zero for some nodes then a non–complete traversal may be optimal because the nodes for which

$Q(A)$ is zero need not be searched. However, there will be a complete serial traversal that will have the same cost (as defined in the following paragraph) as the optimal traversal. For instance, assume that s is an optimal non-complete serial traversal which goes through nodes for which the value of Q is non–zero and s leaves out the nodes with zero–probability. Such a traversal can be made complete by extending it with a walk that covers the nodes with zero–probability, which does not increase its cost according to (3) because $Q(A)$ will be zero when A contains the zero–probability nodes. Thus each optimal traversal has at least one corresponding optimal complete traversal, and no generality is lost by considering only the complete ones. In the following discussion, unless explicitly stated, a serial traversal or a walk is assumed to be complete.

The cost incurred in finding a resource depends on, Q, the probability distribution for the well–informed set and the serial traversal used. If s is the serial traversal used and the resource is found at node v, the searcher traverses the links up to the first visit of v. We use s_v to denote the subtraversal of s that ends at the first visit of v. Thus, if the resource is found at node v, the cost of finding it is the length of s_v which is the sum of all edges in s_v. If we denote s's length by $\ell(s)$, and $Q_s(v)$ is the probability that the resource is found at node v, the cost of finding the resource using s, C_s^Q, can be written as,

$$C_s^Q = \sum_{v \in V} \ell(s_v) Q_s(v) \tag{1}$$

$Q_s(v)$ can be easily derived from Q,

$$Q_s(v) = \sum_{A \cap V(s_v) = \{v\}} Q(A) \tag{2}$$

Alternatively we can write,

$$C_s^Q = \sum_{A \subset V} H_s^A Q(A) \tag{3}$$

where H_s^A is the cost of s conditioned on the fact that $A \subset V$ ($|A| = n$) is the well–informed set.

From the cost formulas derived above, it can be seen that our decision to consider serial traversals only excludes walks which are not optimal. This is proved in the following lemma and hence, when our goal is to minimize the communication cost of finding resources, it is only necessary to consider serial traversals.

Lemma 1 *Let $w = (v_0, v_1, \ldots, v_l)$, $l \geq 2$, be a complete walk that is not a serial traversal. Then, there is another complete walk w', such that $C_{w'}^Q \leq C_w^Q$.*

Proof: The cost defined for a serial traversal also applies to a complete walk. If w violates condition (b) in definition 1, v_l is not a first visit. For $w' = (v_0, \ldots, v_{l-1})$ it is clear that w' is complete and $C_{w'}^Q = C_w^Q$. If w violates condition (a), then there are $i < j$, such that $w_1 = (v_i, \ldots, v_j)$ is not a shortest path and none of the node v_m for $i < m < j$ is a first visit. We consider a shorest path $w_2 = (v_i', \ldots, v_j')$ where $v_i' = v_i$ and $v_j' = v_j$, and we construct w' by substituting w_2 in place of w_1. It can be seen that this w' is such that $C_{w'}^Q \leq C_w^Q$ (since w is complete so is w'). ■

2.2 Resource Finding Algorithms

A resource finding algorithm is a "scheduler" which, given a probability distribution for the well–informed set of a resource, produces a serial traversal to be used as the plan of the search. In general for each starting node, v, a random complete serial traversal may be chosen based on a probability distribution $R_v : \mathcal{C}(G, v) \rightarrow [0, 1]$. Thus we will identify a network–wide resource finding algorithm with the family of distributions $R = \{R_v\}_{v \in V}$. $R_v(s)$ is the probability that the algorithm produces serial traversal $s \in \mathcal{C}(G, v)$, when starting the search at node v. A *deterministic* algorithm is a special case and will always choose the same serial traversal. That is, the distributions R_v will have value 1 for a particular serial traversal and zero for all others.

We use $J(R, Q, v)$ to denote the average cost of a resource finding algorithm starting at node v that uses R to choose among serial traversals and Q is the probability distribution for the well–informed set. We get

$$J(R, Q, v) = \sum_{s \in \mathcal{C}(G,v)} C_s^Q R_v(s) \tag{4}$$

A resource can migrate while another node is trying to locate it and hence the membership of its well–informed set may change. When a particular serial traversal is being used, the nodes already visited by it could comprise the new well–informed set and hence the resource will not be found even when the serial traversal is complete. However, to ensure correctness, the searcher node can augment the scheme. For example, the search can be restarted after some time when the change in the well–informed set has taken effect. Since we expect migration to be infrequent compared to the rate of requests for finding a resource in a real system, such a situation would be rare and it will not have an impact on the average cost of resource finding.

3 Non–Informed Search

In a distributed system, a node may not have any information about the current location of a remote resource. As discussed earlier, this could happen either when the node does not monitor information about all remote resources or the information is allowed to be temporarily incorrect. We will investigate the resource finding problem when the searcher knows only the number of references, n. The probability distribution indicating the likelihood of a particular set of nodes being the well–informed set, Q, is not known. In such a system, we are interested in estimates of the average cost incurred by resource finding algorithms.

3.1 Complete Networks

We study the problem for complete networks (there is a communication link between every pair of nodes) when $c(e) = 1$ for all $e \in E$. We will use K_N to denote a complete network

with N nodes. Thus our cost measure actually accounts for the number of messages used to locate the resource. The cost of an optimal serial traversal in a complete network will constitute a lower bound on the cost of a serial traversal in any of its subnetworks with the same number of nodes. This is so because the set of complete traversals in a subnetwork is a subset of the set of complete traversals in the complete network.

In a complete network, any node can be reached directly from any other node. If all links have the same cost, no serial traversal will ever visit the same node twice. In other words, the set of all complete traversals, $\mathcal{C}(G, v)$, is just the set of all $(N-1)!$ permutations of nodes in V in which v is the first node.

For an uninformed searcher, any resource finding algorithm will make its selection of the serial traversal independent of the resource for which the search is being undertaken. It will only depend on the network and the node starting the search. This is similar to the model presented by Mullender and Vitanyi in [3], where the set of nodes to be queried by a searcher node depends only on the node and not the resource being requested. It is proved in [3] that the algorithm has a lower bound of $2\sqrt{N}$ messages per location operation on the average (including the messages needed to update the resource references). This lower bound is reached in a complete network in which the references are distributed in a particular way forming what the authors call a fully distributed scheme.

We now demonstrate a resource finding algorithm based on serial traversals that has this lower bound when nodes are non–informed, and the bound is reached in complete networks with a unit cost function when n, the number of references, is set to a certain value. Since Q is not known, the searcher has no information which can be used to distinguish one serial traversal from another. Thus, it should choose a serial traversal randomly. In particular, we will use algorithm R which chooses each complete serial traversal with the same probability ($R_v(s) = \frac{1}{|\mathcal{C}(G,v)|}$ for all $s \in \mathcal{C}(G, v)$). We call R the *uniform algorithm.* We will show that R has the desirable property that, independent of the unknown underlying distribution Q, its average cost is $\frac{N}{n+1}$. Furthermore, R is distributed because any node will be equally likely to participate in finding a resource since each serial traversal is chosen with the same probability.

Theorem 1 *For $G(V, E) = K_N$ and $c(e) = 1$ for all $e \in E$, the location cost when R_v is uniform; i.e., $R_v(s) = \frac{1}{|\mathcal{C}(G,v)|}$ $\forall s \in \mathcal{C}(G, v)$, is independent of Q and is given by*

$$J(R, Q, v) = \frac{N}{n+1}$$

Proof: Let N_k be the number of serial traversals which, for a given $A \subset V$ ($|A| = n$), have a cost $H_s^A = k$. Such traversals will not have any of their first $k-1$ nodes in A, and there are $\binom{N-1-n}{k-1}$ possible choices for those nodes. On the other hand different orderings of those nodes will produce different traversals. There are $(k-1)!$ ways of ordering those $(k-1)$ nodes, thus, in total there are $\binom{N-1-n}{k-1}(k-1)!$ different ways to select the sequence of the first $(k-1)$ nodes of a serial traversal with the desired property. Next we have n different choices for the node v_k (which will contain the first reference found along the traversal).

Finally, we have $(N-1-k)!$ different ways to order the remaining $(N-1-k)$ nodes in the traversal. Thus, the total number of traversals with the property is given by,

$$N_k = \binom{N-1-n}{k-1}(k-1)!n(N-1-k)!$$

We have that $|\mathcal{C}(G,v)| = (N-1)!$. Using (4) we have

$$\begin{aligned}
J(R,Q,v) &= \sum_{s\in\mathcal{C}(G,v)} C_s^Q R_v(s) = \frac{1}{(N-1)!}\sum_{s\in\mathcal{C}(G,v)}\sum_{A\subset V} H_s^A Q(A) \\
&= \frac{1}{(N-1)!}\sum_{A\subset V} Q(A)\sum_{k=1}^{N-n}\sum_{\substack{s\\ H_s^A=k}} H_s^A = \frac{1}{(N-1)!}\sum_{A\subset V} Q(A)\sum_{k=1}^{N-n} kN_k \\
&= \frac{1}{(N-1)!}\sum_{k=1}^{N-n} k\binom{N-1-n}{k-1}(k-1)!n(N-1-k)! \\
&= \frac{n}{(N-1)!}\sum_{k=1}^{N-n}\frac{(N-1-n)!}{(k-1)!(N-n-k)!}k(k-1)!(N-1-k)! \\
&= \frac{(N-1-n)!n}{(N-1)!}\sum_{k=1}^{N-n} k(n-1)!\frac{(N-1-k)!}{(n-1)!(N-n-k)!} \\
&= \frac{1}{\binom{N-1}{n}}\sum_{k=1}^{N-n} k\binom{N-1-k}{n-1}
\end{aligned}$$

To proceed with the summation let us drop the factor $\frac{1}{\binom{N-1}{n}}$ for the moment. We will consider it later. We first perform a change in the summation index, substituting k for j, where $j = N-n-k$,

$$\begin{aligned}
\sum_{k=1}^{N-n} k\binom{N-1-k}{n-1} &= \sum_{j=0}^{N-n-1}(N-n-j)\binom{n-1+j}{j} \\
&= (N-n)\sum_{j=0}^{N-n-1}\binom{n-1+j}{j} - \sum_{j=1}^{N-n-1} j\binom{n-1+j}{j}
\end{aligned}$$

The desired result follows by applying the properties

$$j\binom{m+j}{j} = (m+1)\binom{m+j}{j-1}$$

and,

$$\sum_{j=0}^{m}\binom{n+j}{j} = \binom{m+n+1}{m}$$

■

If an algorithm does not choose each traversal with the same probability as R does, there will be instances of the probability distribution Q for which the average search cost

will be higher than the one given above. As an example, assume that the searcher uses a deterministic algorithm R, for which $R_v(s) = 1$ for a particular traversal, $s \in C(G, v)$. Thus s would be used each time the location of a resource needs to be found at node v. Let $G = K_5, n = 1$ and let $s = (v_0, v_1, v_2, v_3, v_4)$ be such that $R_{v_0}(s) = 1$. If $Q(\{v_4\}) = 1$, then $J(R, Q, v_0) = 4$ instead of 2.5 as given by lemma 1. Thus, to make sure that the average cost of finding a resource is $\frac{N}{n+1}$, R should be uniform ($R_v(s) = R_v(s')$ for all $s, s' \in C(G, v)$).

From the cost formulas derived above, it follows that when the number of references stored for a resource is equal to $N - 1$, then the cost of finding it reduces to 1 (clearly the absolute minimum when it is not found locally). However, distributing the references in the network to n nodes will incur its own cost when the resource changes location. In order to distribute n references to a given resource, $n - 1$ nodes have to be notified (the node where the resource moves, already has a reference to it). In complete networks, this requires that $n-1$ links be traversed. Thus the total cost per location operation would have to account for the cost of distributing the references as well. A straightforward way to do this is to divide the cost of distributing n references among all the location operations that take place between two consecutive update operations (updates are done when the resource migrates to a new node). Considering the rates of update and location requests, μ and λ, respectively, the number of location operations between two consecutive update operations would be given by $\frac{\lambda}{\mu}$. Thus we would have to add $\frac{\mu}{\lambda}(n-1)$ to the cost of each location operation, obtaining the following formula for the total cost per location operation,

$$T = \frac{N}{n+1} + \frac{\mu}{\lambda}(n-1) \tag{5}$$

The following theorem shows that $T = \Omega(\sqrt{\frac{\mu}{\lambda}N})$ when $\frac{4}{N} \leq \frac{\lambda}{\mu} \leq N$.

Theorem 2 *If $\frac{4}{N} \leq \frac{\lambda}{\mu} \leq N$, then the number of references that minimizes T is $n_{\min} = \sqrt{\frac{\lambda}{\mu}N} - 1$, and the minimum total cost would be $T_{\min} = 2(\sqrt{\frac{\mu}{\lambda}N} - \frac{\mu}{\lambda})$.*

Proof: Straightforward by taking the derivative of the cost formula and equating it to zero. ∎

In the above theorem we have considered only cases in which $\frac{4}{N} \leq \frac{\lambda}{\mu} \leq N$, this is due to the fact that for all other cases, the minimum of the cost formula is attained for values of n less than 1 or greater than $N - 1$. In those cases, clearly, the optimum strategy would be to keep just one reference or broadcast a reference to the resource to all nodes in the network respectively.

Theorem 3 *The following properties hold,*

a) If $\frac{\lambda}{\mu} < \frac{4}{N}$, then $n_{\min} = 1$ and $T_{\min} = \frac{N}{2}$

b) If $\frac{\lambda}{\mu} > N$, then $n_{\min} = N$ and $T_{\min} = (N-1)\frac{\mu}{\lambda} < 1$.

3.2 Extensions to General Networks

The results of the previous section are for complete networks. For arbitrary networks, we can only provide a lower bound, that is, for a general $G(V,E)$, $T_{\min}(G) \geq T_{\min}(K_{|V|})$. For non–complete networks it is not clear that all possible serial traversals have to be considered with equal probability. The difficulty is due to the fact that the topology of the network imposes restrictions on how to navigate in the network.

We can provide an upper bound on the cost of finding a resource using serial search by giving a particular algorithm and deriving its cost. The cost of an optimal algorithm will be smaller or equal to the cost of the above algorithm. In the following theorem we present an algorithm which does not choose every traversal with the same probability, and whose cost is bounded by a quantity which will be shown to be tight for general networks.

Theorem 4 *Let $G(V,E,c)$ be a distributed system (c is constant and equal to 1), then there is an algorithm R for which $J(R,Q,v) \leq N - n$.*

Proof: Given an arbitrary connected graph $G(V,E)$, we can find a spanning tree for it, which is rooted at v and has $N-1$ edges. Let $w = (v = v_0, \ldots, v_u, \ldots, v_{2(N-1)} = v)$ be a depth–first traversal of the tree, where v_u is the last first visit in the traversal, and $(v_u, \ldots, v_{2(N-1)})$ is the shortest path from v_u to v. Based on the traversal w, we can traverse the tree in two different ways: we can start at v_0 and proceed from v_i to v_{i+1}, or we can start at $v_{2(N-1)}$ and proceed from v_i to v_{i-1}. We can associate serial traversal $s_1 = (v_0, \ldots, v_u)$ with the first form of traversing the tree and $s_2 = (v_{2(N-1)}, \ldots, v_d)$ with the second, where v_d is, similarly to v_u, the last first visited node in the traversal $w' = (v_{2(N-1)}, \ldots, v_d, \ldots, v_0)$ (i.e., the reversed traversal of w).

We consider the algorithm R, defined by $R_v(s_1) = R_v(s_2) = \frac{1}{2}$. The maximum number of edges to be traversed is $2(N-1)-1$. Let us assume that node v_k is the first visited node in s_1 with a resource reference, then when following traversal s_1, we will incur a cost of k. On the other hand, if the subtree rooted at v_k has m elements, the last time node v_k will be visited is $v_{k+2(m-1)}$. The question we ask now is what is the maximum cost we could incur if we followed the traversal s_2. If all references are inside the subtree rooted at v_k, then the first node containing a reference which would be found by following s_2 is $v_{k+2(m-1)}$. If there is a node, v_i, outside the tree, that node will be found before $v_{k+2(m-1)}$ when following s_2. Thus, if all the references are in the subtree rooted at v_k, the search proceeding along traversal s_2 would reach its maximum cost. This cost would be $2(N-1)-(k+2(m-1))$. By our assumptions we have that $m \geq n$, the number of references to the resource. Thus the above cost will reach a maximum when $m = n$. In other words, if A is the well–informed set, and traversal s_1 incurs the cost $H^A_{s_1}$, the cost incurred by following traversal s_2 will be at most $2(N-1)-(H^A_{s_1}+2(n-1))$. Thus

$$\begin{aligned} J(R,Q,v) &= C^Q_{s_1} R_v(s_1) + C^Q_{s_2} R_v(s_2) \\ &= \frac{1}{2}\left(\sum_{A \subset V, |A|=n} (H^A_{s_1} + H^A_{s_2}) Q(A) \right) \end{aligned}$$

$$\leq \frac{1}{2}\left(\sum_{A\subset V,|A|=n}(H_{s_1}^A + 2(N-1) - (H_{s_1}^A + 2(n-1)))Q(A)\right)$$
$$= (N-n)$$

■

Theorem 4 provides a tight upper bound for general networks when Q is unknown (i.e., it cannot be lowered without changing the assumptions of the theorem). For example, for all path networks in which the search starts at one of the extremes and the set A contains the n nodes farthest away from the starting node, the minimum cost will be $N-n$.

4 Optimal Resource Finding for Informed Searchers

In this section we investigate the problem of finding the optimal way to conduct a serial search in a general network when the searcher knows the distribution Q. Since the cost of resource finding will be high for uninformed searchers, nodes can reduce this cost by maintaining information from which Q can be estimated. As stated earlier, such information may include locality of resource access as well as usage and migration patterns.

A resource finding algorithm R will be optimal when $J(R,Q,v)$ is minimal for each v. It can be readily seen that for each $v \in V$, there is a certain serial traversal (not necessarily unique), $s(v) \in \mathcal{C}(G,v)$, such that the algorithm R defined by $R_v(s(v)) = 1$ is an optimal one. This will happen when $s(v)$ is such that $C_{s(v)}^Q$ is minimal for all $s \in \mathcal{C}(G,v)$. In other words, there is an optimal resource finding algorithm which is deterministic. For a given node v, we will thus be interested in the problem of finding the complete serial traversal, $s \in \mathcal{C}(G,v)$ for which C_s^Q is minimal. We will say that such a serial traversal is optimal.

We will restrict the model to those networks in which $n = 1$. In that case $Q(\{v\})$ becomes the probability that the resource resides at node v, and we will use the notation $\Pi(v)$ instead. For this particular case it can be seen that the cost formula presented in (1) reduces to the following expression.

$$C_s^\Pi = \sum_{v\in V} \ell(s_v)\Pi(v) \tag{6}$$

We now consider the problem of finding an optimal serial traversal for various types of networks when Π is given. We present several results concerning the time complexity of the optimal serial traversal problem by showing that the optimization problem (as it will be presented later) is *NP-hard* for general graphs. This result was also shown by Trummel and Weisinger [15] in a different formulation. We will show the additional result that holds for complete graphs with non-unit edge costs. Notice that the NP-hardness results obtained for our special case are also applicable to the general problem in which the number of references is not fixed. We will first prove the following lemma which will be used in the NP-hardness proofs.

Lemma 2 *Let $G(V,E,c)$ be a distributed system, and let Π be the uniform distribution on $V-\{v_0\}$, $\Pi(v)=\frac{1}{N-1}$ for all $v \neq v_0$, where N is the number of nodes in the system. Let $M=\min_{e\in E}\{c(e)\}$, then,*

(i) *If $s=(v_0,v_1,\ldots,v_{N-1})$ is a complete traversal such that $c(\{v_i,v_{i+1}\})=M$ for all i, $0\leq i<N-1$, then $C_s^{\Pi}=M\frac{N}{2}$ and s is optimal.*

(ii) *Consider the graph $G'(V,E')$, where $E'=\{\{v,v'\}|\{v,v'\}\in E$ and $c(\{v,v'\})=M\}$. Then a traversal $s\in C(G,v_0)$ has cost $C_s^{\Pi}=M\frac{N}{2}$ if and only if s is a hamiltonian path starting at v_0 in G', and such a traversal is optimal.*

Proof: Since $C_s^{\Pi}=\sum_{v\in V}\ell(s_v)\Pi(v)$ and $\ell(s_{v_i})=Mi$, $C_s^{\Pi}=\sum_{i=0}^{N-1}\frac{Mi}{N-1}=M\frac{N}{2}$. To show that s is optimal, let $s'=(v_0=v'_0,\ldots,v'_l)$ be another complete serial traversal. If s' satisfies the conditions of s then its cost will be the same as the cost of s. Otherwise, either $l\geq N$ or $c(\{v'_i,v'_{i+1}\})>M$ for some i. Then $C_{s'}^{\Pi}>M\frac{N}{2}$, which proves that s is optimal.

In part (ii), if $C_s^{\Pi}=M\frac{N}{2}$ then s will have to satisfy the conditions of part (i), which implies that s is optimal and it clearly is a hamiltonian path in G' (and in G itself). On the other hand, if s is a hamiltonian path in G' and the cost of each link is M so C_s^{Π} is $M\frac{N}{2}$.

■

4.1 General Networks

In general networks, the optimal traversal problem is *NP-hard* even in the simple case when Π is uniform and link costs are the same (c is constant).

Theorem 5 *The problem of finding an optimal serial traversal starting at a particular node of a distributed system $G(V,E,c)$, with c the constant function and equal to 1, is NP–hard (see [15]).*

Proof: We will prove this result by reducing the hamiltonian path problem to our problem. Let $G(V,E),v_0\in V$ be an instance of the hamiltonian path problem. Then we construct the following instance of the optimal traversal problem: $G(V,E,c),v\in V,\Pi$ with $c=1$ and Π as the uniform distribution on $V-\{v_0\}$. Let $G'(V,E')$ be such that $E'=\{\{v,v'\}|\{v,v'\}\in E$ and $c(\{v,v'\})=1\}$. We would have $E'=E$ (all edges have the same cost) and thus, $G'(V,E')=G(V,E)$, and from part (ii) of lemma 2, G will have an optimal traversal with cost $\frac{N}{2}$ if and only if G has a hamiltonian path. This, together with the fact that the reduction presented can be clearly carried out in polynomial time completes the proof. ■

4.2 Complete Networks

In complete networks, if the link cost function is a constant ($c(e)=M$ for all $e\in E$), it can be easily seen that an optimal traversal will visit nodes in decreasing order for the value of the probability distribution Π. Since nodes can be sorted according to decreasing order of $\Pi(v)$ in $O(NlogN)$ time, the optimal traversal can be found efficiently in complete networks

when the edge cost function c is a constant. If c is not constant, however, the problem becomes *NP-hard*, even for the simple case in which the distribution Π is uniform. Thus, a small increase in the complexity of the problem renders it computationally intractable.

Theorem 6 *The problem of finding the optimal serial traversal in a distributed system, $G(V, E, c)$, where $G(V, E) = K_N$ and $c(e) = 1$ or 2 for all $e \in E$, and where Π is the uniform distribution, is NP–hard.*

Proof: We will prove the result by reducing the hamiltonian path problem to our problem. Let $G(V, E), v_0$, where $v_0 \in V$, be an instance of the hamiltonian path problem starting at v_0. We will transform it to the problem $G'(V, E', c), v_0$, where $G'(V, E') = K_N$, and c is defined as

$$c(\{v, v'\}) = \begin{cases} 1 & \text{if } \{v, v'\} \in E \\ 2 & \text{otherwise} \end{cases}$$

The edges in G' with cost 1 correspond to edges in G. Thus, the set $E'' = \{\{w, w'\} | c(\{w, w'\}) = 1\}$ is exactly E and, thus, $G''(V, E'') = G(V, E)$. Using Π as the uniform distribution on $V - \{v_0\}$, we know from part (ii) of lemma 2 that an optimal traversal in G' will have cost $\frac{N}{2}$ if and only if G has a hamiltonian path. On the other hand the above reduction can obviously be carried out in polynomial time, from which the theorem follows. ■

4.3 Tree Networks

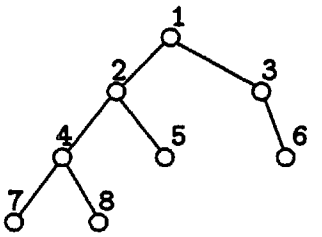


Figure 1: Example tree

It seems clear from the above that an algorithm to solve the optimal serial traversal problem is computationally intractable for arbitrary graphs. It may, however, be possible to identify a subclass of graphs for which there exists a more efficient algorithm. A heuristic can then be used for general graphs by applying the algorithm to a subgraph in the class which is efficiently solvable.

In this section, we will study the class of tree structures. We first consider trees in which costs of all edges are the same and the distribution Π is uniform. In this case, an optimal traversal can be found efficiently because we will show that any depth–first traversal is optimal. When the edge costs and Π are not uniform, we present an algorithm which runs in polynomial time for a certain class of trees with a restricted number of what we call *frontiers* (to be defined later). Examples of such subclasses of trees are line graphs and

star graphs. Furthermore, we will show how to use the algorithm to solve the problem in a bidirectional ring network. The algorithm presented in this section to find an optimal traversal in trees is based on dynamic programming.

We designate the node where search starts in a tree network as the root of the tree which will be represented by the letter r. We will assume node 1 to be the root in the tree in figure 1 in our examples. We first define some concepts that will be used in the proofs for tree networks.

Let $G(V, E)$ be a graph, and let s_1 and s_2 be two possibly non–complete serial traversals, $s_1 = (v_0, \ldots, v_m)$ and $s_2 = (u_0, \ldots, u_p)$. If $v_m = u_0$ then we say that s_2 is *composable* with s_1 and we will define their composition, $s_1 \cdot s_2 = (v_0, \ldots, v_m = u_0, \ldots, u_p)$, which is the sequence resulting from appending s_2 to s_1, without repeating u_0. If $s_1 \cdot s_2$ is also a serial traversal then we say that s_2 is *compatible* with s_1. An example of two composable serial traversals in the tree of figure 1 is the following: $s_1 = (1, 2, 1, 3)$ $s_2 = (3, 6, 3, 1, 2, 4)$, where the composition $s_1 \cdot s_2 = (1, 2, 1, 3, 6, 3, 1, 2, 4)$. Furthermore, s_2 is compatible with s_1 since $s_1 \cdot s_2$ is also a serial traversal. As an example of composable but not compatible serial traversals, consider traversals s_1 and s_2 in figure 1 where $s_1 = (1, 2, 1, 3)$ and $s_2 = (3, 1, 2)$. The composition $s = s_1 \cdot s_2 = (1, 2, 1, 3, 1, 2)$ is not a serial traversal because the last node of s is not a first visit and hence s_2 is not compatible with s_1.

In a tree, there is a unique simple path between any pair of nodes. We will denote by $L_{u,v}$ the unique simple path joining nodes u and v. The following lemma shows that any serial traversal (not necessarily complete) is the composition of a serial subtraversal and a simple path.

Lemma 3 *Let $s = (v_0, \ldots, v_l = v)$ such that $l > 1$. There is a serial traversal s' such that $s = s' \cdot L_{v',v}$ and $V(s') = V(s) - \{v\}$.*

Proof: s can be written as follows, $s = (v_0, \ldots, v_{m-1}, v_m, \ldots, v_l = v)$, where v_m is a first visit and for all i such that $m < i < l$, v_i is not a first visit. Let $s' = (v_0, \ldots, v_m)$. s' is also a serial traversal because v_m is a first visit and it must satisfy condition (b) of definition 1 because otherwise s will not be a serial traversal. On the other hand, from condition (b) of definition 1, $(v_m, \ldots, v)$ is a shortest path between v_m and v, that is, in our case it is the simple path $L_{v_m,v}$. It is also clear from the above that $V(s') = V(s) - \{v\}$, thus letting $v' = v_m$ we have proved the lemma. ∎

4.3.1 Tree Networks with Uniform Cost and Probability Distribution

In the special case when the distribution is uniform and all the edge costs in the tree network are equal, it is possible to give an efficient algorithm to find the optimal serial traversal. In fact, traversals producing the minimal cost will be *depth–first* traversals, which first–visit the nodes in an order that is a possible search order for depth–first search. The main property of a depth–first traversal is that once the traversal leaves a subtree rooted at v, it never again visits node v.

When the root is fixed in a tree, we can associate each link in it with one of its nodes (with the descendant node). If s is a depth–first traversal of the tree ending at node u (complete traversals are assumed unless indicated otherwise), then the following will be true of s

- The link leading to a node v not in $L_{r,u}$ will be traversed exactly twice since the traversal has to get to v and then out of v to reach u. There are $(N-1) - \ell(L_{u,r})$ such links since s visits all nodes.
- The link leading to a node in $L_{r,u}$ is traversed exactly once (to get to u). There are $\ell(L_{u,r})$ such links.

Thus the length of s, $\ell(s)$, would satisfy

$$\ell(s) = 2[(N-1) - \ell(L_{r,u})] + \ell(L_{r,u}) = 2(N-1) - \ell(L_{r,u}) \tag{7}$$

If s' is a complete serial traversal which may not visit the nodes in a depth–first search order and it also ends at node u of tree T then the link leading to a node v not in $L_{r,u}$ will be traversed at least twice (the traversal has to get to v and then out of v to reach u but its visit order may require it to pass through v more times). Therefore,

$$\ell(s') \geq 2[(N-1) - \ell(L_{r,u})] + \ell(L_{r,u}) = 2(N-1) - \ell(L_{r,u}) \tag{8}$$

Thus, depth–first traversals have minimum length once the ending node is fixed.

Lemma 4 *If T is a tree with $c(e) = 1$ for all e, and s and s' are two depth–first traversals of T then $C_s^{\Pi} = C_{s'}^{\Pi}$.*

Proof: Since Π is assumed to be uniform, $C_s^{\Pi} = \sum_{v \in V} \ell(s_v)\Pi(v) = \frac{1}{N-1}\sum_{v \in V} \ell(s_v)$. Similarly, $C_{s'}^{\Pi} = \frac{1}{N-1}\sum_{v \in V} \ell(s'_v)$. Therefore, we only need to prove that $\sum_{v \in V} \ell(s_v) = \sum_{v \in V} \ell(s'_v)$. We denote $\sum_{v \in V} \ell(s_v)$ by σ_s and hence need to prove that $\sigma_s = \sigma_{s'}$. We will prove it by induction on the size of the tree. For $N = 1$ there is only one tree and only one traversal, thus the property holds. Let's now assume the property holds for all trees with cardinality up to $N-1$ for $N > 1$. We will now prove that it holds for N. Let T be a tree with N nodes, rooted at r. Let r_1 to r_k be the descendant nodes of r. Let T_i denote the subtree rooted at r_i. Finally, let N_i denote the size of subtree T_i. Let us consider a depth–first traversal s of tree T. Such traversal will visit each one of the above trees in turn. Due to the fact that s is a depth–first traversal, once it starts traversing tree T_i it will continue with T_i until it has visited all its nodes. Without loss of generality, let's assume that s traverses the above subtrees in the order given by the subindices, that is it first visits T_1, then T_2 and so on, leaving T_k last. We can decompose s as follows:

$$s = (r, r_1) \cdot s_1 \cdot L_{u_1,r_1} \cdot (r_1, r, r_2) \cdot s_2 \cdot L_{u_2,r_2} \cdot (r_2, r, r_3) \cdot \ldots s_{k-1} \cdot L_{u_{k-1},r_{k-1}} \cdot (r_{k-1}, r, r_k) \cdot s_k$$

where each s_i is a depth–first traversal of subtree T_i ending at u_i. Let $v \in V(T_i)$. Then we can write s_v as

$$s_v = (r, r_1) \cdot s_1 \cdot L_{u_1,r_1} \cdot \ldots (r_{i-1}, r, r_i)(s_i)_v$$

Thus $\ell(s_v)$ would be given by

$$\ell(s_v) = 1 + \ell(s_1) + \ell(L_{u_1,r_1}) + 2 + \ldots + \ell(s_{i-1}) + \ell(L_{u_{i-1},r_{i-1}}) + 2 + \ell((s_i)_v)$$

Notice that in the above s_j are depth–first traversals, thus using 7 we can write

$$\ell(s_v) = 1 + 2(N_1 - 1) + 2 + \ldots + 2(N_{i-1} - 1) + 2 + \ell((s_i)_v)$$

which would finally result in

$$\ell(s_v) = 1 + 2\sum_{j=1}^{i-1} N_j + \ell((s_i)_v)$$

Thus we can write σ_s as

$$\begin{aligned} \sigma_s &= \textstyle\sum_{i=1}^{k} \sum_{v \in V(s_i)} (1 + 2\sum_{j=1}^{i-1} N_j + \ell((s_i)_v) \\ &= \textstyle\sum_{i=1}^{k} (N_i + 2N_i \sum_{j=1}^{i-1} N_j + \sigma_{s_i}) \end{aligned}$$

which can be rewritten as

$$\sigma_s = N - 1 + (N-1)^2 - \sum_{i=1}^{k} (N_i)^2 + \sum_{i=1}^{k} \sigma_{s_i}$$

Following an identical argument, we can show that

$$\sigma_{s'} = N - 1 + (N-1)^2 - \sum_{i=1}^{k} (N_i)^2 + \sum_{i=1}^{k} \sigma_{s'_i}$$

By the induction hypothesis, $\sigma_{s_i} = \sigma_{s'_i}$, thus $\sigma_s = \sigma_{s'}$ and $C_s^{\Pi} = C_{s'}^{\Pi}$ follows.

■

Theorem 7 *If Π is a uniform distribution over $V - \{r\}$ and $c(e) = 1$ for all e, then any depth–first serial traversal of the tree is optimal.*

Proof: From lemma 4 we know that all depth–first serial traversals have the same cost. We only have to prove that the cost of a serial traversal is greater than or equal to the cost of a depth–first serial traversal. The cost of a serial traversal would be given by

$$C_s^{\Pi} = \sum_{v \in V} \ell(s_v)\Pi(v) = \frac{1}{N-1} \sum_{v \in V} \ell(s_v)$$

Again, let us denote $\sum_{v \in V} \ell(s_v)$ by σ_s. Let C_T be the cost of any depth–first serial traversal for tree T and let σ_T be the value of $\sum_{v \in V} \ell(s_v)$ for any such traversal s (since the cost is same, so is the value of σ). We will prove that for an arbitrary serial traversal s' of T, $C_{s'}^{\Pi} \geq C_T$ by showing that $\sigma_{s'} \geq \sigma_T$. We will proceed by using induction. The basis is trivial, a depth–first traversal is minimal for a tree with just one node. Assume that it is still true for all trees with size less than or equal to $N-1$ ($N > 1$), we will now prove it for

N. Let s be an arbitrary complete serial traversal of T ending at u. It is easy to see that (see figure 2)

$$s = s_1 \cdot L_{u_1,u_2} \cdot L_{u_2,u}$$

where u_1 is also a leaf node of T and s_1 is a serial traversal ending at u_1, and where $V(L_{u_1,u_2}) \subset V(s_1)$, such that if $v \in V(L_{u_2,u}) - \{u_2\}$ then v is a first visit in s. It is easy to see that all nodes v in $L_{u_2,u}$ (except possibly u_2) have at most one direct descendant. Thus u_1 is the last leaf node which is a first visit in s besides u. Let T_1 be the tree resulting from T after the removal of $V(L_{u_2,u}) - \{u_2\}$. Then s_1 is a complete serial traversal of T_1. We would thus have

$$\begin{aligned} \sigma_s &= \sigma_{s_1} + \sum_{v \in V(L_{u_2,u})-\{u_2\}} \ell(s_v) \\ &= \sigma_{s_1} + \sum_{i=0}^{\ell(L_{u_2,u})-1} (\ell(s) - i) \end{aligned}$$

Let s' be a depth–first serial traversal ending at u. Then we can also write

$$s' = s'_1 \cdot L_{u_3,u_2} \cdot L_{u_2,u}$$

Let T_2 be the subtree rooted at u_2 formed by all descendants of u_2 except $V(L_{u_2,u}) - \{u_2\}$. Then u_3 would be a leaf node of T_2 and s'_1 would also be a depth–first traversal of T_1. Thus,

$$\sigma_{s'} = \sigma_{s'_1} + \sum_{i=0}^{\ell(L_{u_2,u})-1} (\ell(s') - i)$$

By the induction hypothesis $\sigma_{s_1} \geq \sigma_{s'_1}$, and from 7 and 8, $\ell(s) \geq \ell(s')$. Thus we get $\sigma_s \geq \sigma_{s'} = \sigma_T$, which completes the proof. ■

Notice that although in the above lemmas and in the theorem we have assumed that $c(e) = 1$, the results also hold for any other constant value for $c(e)$.

4.3.2 General Tree Networks

In a general tree network, the order in which a serial traversal visits the nodes depends on the link costs and the distribution Π. A serial traversal starts at the root and expands the search by visiting nodes until the resource is found. At any point, the set of nodes visited by the traversal will contain the root node and it is easy to see that the subgraph defined by the visited nodes will be connected. In fact, this subgraph will be a subtree of the tree network. The next definition formalizes some properties of these subtrees which are used in the development of the algorithm which finds an optimal serial traversal.

Definition 2 *Given a rooted tree $G(V,E)$ with root $r \in V$, we say that $B \subset V$ is a border of G if for all $v, w \in B$, $v \notin V(L_{r,w})$. A frontier of $G(V,E)$ is an ordered pair $f = (B, v)$ where B is a border and $v \in B$.*

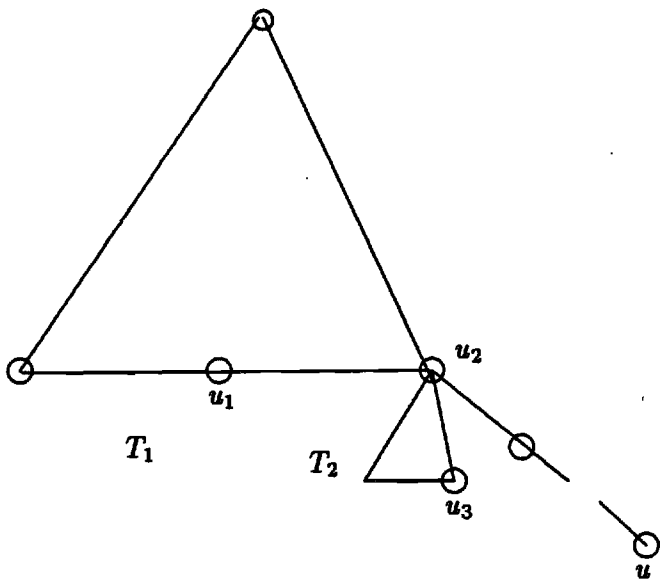


Figure 2: Decomposition of a serial traversal in a tree

In figure 1, the sets $\{2\}$, $\{2,6\}$ and $\{1\}$ are valid examples of borders. The set $\{2,4\}$ however is not a border because $2 \in V(L_{1,4})$. $(\{2,6\},2)$ is an example of a frontier in this tree. In general, if $G(V,E)$ is tree with root r, the set of leaf nodes of any subtree of $G(V,E)$ which is also rooted at r will define a border. Similarly, a border defines a unique subtree of $G(V,E)$ which is rooted at r. This subtree consists of the nodes that are on the path from r to a node in the border. If B is a border and $V(B)$ denotes the set of nodes in the subtree defined by B then $V(B) = \{v | v \in V(L_{r,w}) \text{ and } w \in B\}$.

Since the nodes visited by a serial traversal that starts at r (not necessarily a complete traversal) also induce a subtree of $G(V,E)$ rooted at r, we can associate a border with a serial traversal. If $s = (v_0, \ldots, v_l)$ is a serial traversal (s may not be complete) then v_l is a first visit, which means there will be no $i \neq l$ such that $v_l \in V(L_{r,v_i})$ and hence v_l belongs to the border of s. If we denote by $B(s)$ the border of s (note $V(B(s)) = V(s)$), we can associate a frontier with s, as captured in the following definition.

Definition 3 *We define the following,*

- *Given a serial traversal starting at the root, $s = (v_0, \ldots, v_l)$, we define the frontier associated with the traversal by $F(s) = (B(s), v_l)$.*
- *Given a frontier, f, we define $S(f)$ as the set of serial traversals whose frontier is f; i.e., $S(f) = \{s | s$ starts at r and $F(s) = f\}$.*
- *We also define $O(f,\Pi) = \{s \in S(f) | C_s^\Pi = \min_{s' \in S(f)} C_{s'}^\Pi\}$. Thus, $O(f,\Pi)$ contains minimum cost serial traversals for a given frontier f.*

- *Finally, we define the optimal cost, Ω_f, as , $\Omega_f = C_s^\Pi$ for $s \in O(f,\Pi)$.*

For instance, in figure 1, traversals $s_1 = (1,2,1,3,6)$ and $s_2 = (1,3,1,2,1,3,6)$ are such that $F(s_1) = F(s_2) = f = (\{2,6\},6)$. Furthermore, $S(f) = \{s_1, s_2\}$ and, assuming a uniform Π and unit costs for the links, using (6) we get $C_{s_1}^\Pi = \frac{24}{7}$ and $C_{s_2} = \frac{34}{7}$, thus $O(f,\Pi) = \{s_1\}$.

If $s \in S(f)$ for some $f = (B,v)$, $B \neq \{r\}$, then, by lemma 3, $s = s' \cdot L_{v',v}$ with $V(s') = V(s) - \{v\} = V(B) - \{v\}$. Since $F(s') = f' = (B',v')$ where B' is the border of s', $V(B') = V(s') = V(s) - \{v\}$. When s' is extended by composing $L_{v',v}$ with it to form s, the increase in the average cost due to this extension only depends on the frontier f' and not on s'. To show this, we first need to modify the expression for C_s^Π in (6) when a serial traversal may not be complete. In the case when s may not be complete, the cost incurred when the resource is not found at nodes in $V(s)$ will be $\ell(s)$. The probability of this is $(1 - \sum_{u \in V(s)} \Pi(u))$ and hence C_s^Π can be written as

$$C_s^\Pi = \sum_{u \in V(s)} \ell(s_u)\Pi(u) + \ell(s)(1 - \sum_{u \in V(s)} \Pi(u))$$

Since $s = s' \cdot L_{v',v}$ and $V(s') = V(s) - \{v\}$, we have

$$C_s^\Pi = \sum_{u \in V(s')} \ell(s'_u)\Pi(u) + \ell(s_v)\Pi(v) + \ell(s)(1 - \Pi(v) - \sum_{u \in V(s')} \Pi(u))$$

It can be easily derived from the above and the facts that $V(s') = V(B')$ and $\ell(s) = \ell(s_v) = \ell(s'_{v'}) + \ell(L_{v',v})$

$$C_s^\Pi = C_{s'}^\Pi + \ell(L_{v',v})(1 - \sum_{u \in V(B')} \Pi(u)) \tag{9}$$

Since the second term in the expression for the cost of s does not depend on the particular s' selected, but only on the frontier f', the increase in the average cost only depends on frontier f' when a serial traversal is extended as given in the lemma. This allows us to establish the following result.

Theorem 8 *Let $f = (B,v)$, and let $s \in O(f,\Pi)$ then there is an $f' = (B',v')$ and $s' \in O(f',\Pi)$, such that $s = s' \cdot L_{v',v}$ and $V(s') = V(s) - \{v\}$.*

Proof: Let s' be constructed from s as in the proof of lemma 3. Since s' ends at v', $f' = (B',v')$ where B' is the border of s'. We will prove that such s' belongs to $O(f',\Pi)$ if $s \in O(f,\Pi)$. Assume to the contrary that $s' \notin O(f',\Pi)$, then there is an $s'' \in O(f',\Pi)$ such that $C_{s''}^\Pi < C_{s'}^\Pi$. Let us now consider the following traversal $s''' = s'' \cdot L(v',v)$. By (9) we would have

$$C_{s'''}^\Pi = C_{s''}^\Pi + \ell(L_{v',v})(1 - \sum_{w \in V(B')} \Pi(w)) < C_{s'}^\Pi + \ell(L_{v',v})(1 - \sum_{w \in V(B')} \Pi(w)) = C_s^\Pi$$

But the above cannot be true because s is assumed to be optimal. Thus such an s'' cannot exist and s' itself has to be optimal. ■

Using this theorem, we can now give an algorithm based on dynamic programming to find an optimal serial traversal of a tree. It starts with frontiers for which the set of nodes in subtrees defined by their borders has cardinality 1, until it gets to those with cardinality N. The algorithm also finds an optimal serial traversal for each frontier of the tree using the results of theorem 8. The frontier with minimal cost among those with cardinality N is selected and the serial traversal corresponding to it is the optimal.

In the algorithm to follow, F_i stands for the set of frontiers for which the cardinality of $V(B)$ is i where B is the border of a frontier in F_i, and S_f will represent an optimal traversal for frontier f, i.e., $S_f \in O(f, \Pi)$.

Algorithm 1

Precompute $\ell(L_{v,v'})$ for all pairs $\{v, v'\}$;
$f := (\{r\}, r)$; (* note $F_1 = \{f\}$ *)
$\Omega_f := 0$;
$S_f := (r)$;
for $i := 2$ **to** N **do**
    Find the set F_i, based on F_{i-1};
    for each $f \in F_i$ **do**
        $(B, v) := f$;
        Let $D(B, v)$ be the subset of F_{i-1} whose frontiers
            $f' = (B', v')$ are such that $V(B') = V(B) - \{v\}$;
        $\Omega_f := \min_{f' \in D(B,v)}\{\Omega_{f'} + \ell(L_{v',v})(1 - \Pi(V(B')))\}$;
        $(B_m, v_m) := f_m$; (* the frontier yielding the above minimum. *)
        $S_f := S_{f_m} \cdot L_{v_m,v}$;
    endfor
endfor
return(S_f), where f is a frontier in F_N
    with minimum Ω_f.

The correctness of the algorithm follows from theorem 8, which guarantees that to find an optimal traversal $f = (B, v)$, only the frontiers $f' = (B', v')$, with $V(B') = V(B) - \{v\}$ have to be inspected. The above algorithm represents a general procedure to solve the problem in trees. However, its execution time in the general case will be exponential in the number of nodes since the number of frontiers could be exponential. For particular subclasses of trees, the algorithm can be specialized to increase its efficiency. For instance, when the only nodes with non–zero probability are the leaves of the tree, the only frontiers that have to be considered are those whose border contains only leaf nodes. Similarly, in a star network with b branches and maximum branch length of h, the maximum number of frontiers will be bounded by $(h + 1)^b \cdot b$. Thus for constant b the algorithm will run

in polynomial time. A path graph is a special case of a star with at most two branches. The borders of a path graph consist of at most two nodes, each at a different branch from the root. Thus the total number of frontiers will be bounded by N^2, making algorithm 1 polynomial. Although a bidirectional ring is not a tree, the following lemma shows how algorithm 1 can be adapted for rings.

Lemma 5 *Given a bidirectional ring network $G(V, E, c)$ and a complete serial traversal s for the ring, there is one link (edge) in E which is not traversed by s.*

Proof: Let $s = (v_0, \ldots, v_{l-1}, v_l)$. Each node in a ring network is attached to two different links. In particular, v_l is attached to two links, one of them is $\{v_{l-1}, v_l\}$. If s traverses all the links in the ring, then it will traverse the other link attached to v_l, which means that there is an i such that $0 < i < l$ and $\{v_{i-1}, v_i\}$ is the other link attached to v_l. But this means that $v_l \in \{v_{i-1}, v_i\}$, which implies that v_l is not a first visit, thus contradicting the hypothesis that s is a serial traversal. ■

Thus, if s is an optimal serial traversal and e is one of the unused edges, s would also be an optimal serial traversal for the line graph obtained by eliminating e from E. Thus the previous algorithm can be modified in the following way:

Algorithm 2

for each $e \in E$ consider $G_e(V, E - \{e\}, c)$
    Find an optimal traversal s_e for G_e applying algorithm 1;
endfor
Let $s = s_e$ such that $C_s^{\Pi} = \min_{e' \in E} C_{s_{e'}}^{\Pi}$;
return s

It is possible to make use of algorithm 1 in a general network as a heuristic. For example, a particular path in a network could be selected as the set of name servers, holding references to resources in the network. The algorithm presented here could be used to find a resource reference. As a more concrete example, let us consider bidirectional manhattan networks [16, 17]. In such networks, nodes could transmit references for their resources to all the nodes in their column (row). To locate a resource a node would then apply algorithm 2 to its row (column). This scheme to distribute the references to resources in a Manhattan network is similar to that presented by Mullender and Vitanyi in [3]. Thus, it is possible to apply the results in this section to construct heuristics for graphs other than "polynomial-frontier" trees.

4.3.3 Numerical Examples

We now show how algorithm 1 can be used to find optimal serial traversals in tree networks. Let us consider the tree in figure 1 and assume that the search starts at node 1.

We will use the notation $\underline{\Pi}$ to represent the vector $\underline{\Pi} = (\Pi(2), \ldots, \Pi(8))$. When Π is uniform we get the optimal traversal presented in figure 3–(a). We notice that the optimal traversal is a depth–first serial traversal of the tree as was expected from theorem 7. When $\underline{\Pi} = (0.01, 0.01, 0.01, 0.2425, 0.2425, 0.2425, 0.2425)$, that is, the leaf nodes have a higher probability than the internal nodes, the optimal traversal presented in figure 3-(b) goes first to the subtree containing the largest number of leaf nodes, then proceeding with the other subtree. For the same probability distribution, when we make the costs of the edges $\{4, 7\}$ and $\{4, 8\}$ equal to 2, we get the traversal in figure 3–(c), which decides to leave the traversal of the heavier weights until the end. Finally, for the probability distribution $\underline{\Pi} = (0.5, 0.4, 0.002, 0.08, 0.014, 0.002, 0.002)$, we get the optimal traversal in figure 3-(d). This traversal visits higher probability nodes first.

If we consider a ring consisting of 8 nodes, and we start the search from node 1, when the probability distribution is uniform, the optimal traversal, as shown in figure 4–(a), traverses the ring in one direction without turning back. When the probability is 0.3 for both neighbors of node 1 (nodes 2 and 8), and 0.08 for the other nodes, the optimal traversal, as shown in figure 4–(b), first visits the two highest probability nodes and then proceeds with the rest of them.

5 Concluding Remarks

The average cost incurred by a searcher node which has no knowledge about the location of the resource is high in large networks. In this situation, the cost of serial search is $\Omega(\sqrt{\frac{\mu}{\lambda}N})$ and this bound has also been shown to hold for various other methods (two of the forwarding address protocols [5] which are also serial in nature as well as the scheme in [3]). The conclusion that we draw is that it is necessary to possess more knowledge about the location of a resource to make the process of finding it sufficiently efficient.

We have studied the problem of finding an optimal serial traversal when the searcher has the distribution for the well–informed set at request time. In particular we have looked at the case when the well–informed set is a singleton. The results show that such a problem is NP-hard even for complete graphs in which all links do not have the same cost (in fact, when there are only two different values for the link costs). Thus the existence of an efficient algorithm to find an optimal serial traversal is very unlikely. In the last sections we have shown that the problem can be solved in polynomial time for some classes of trees, and based on it we have shown a polynomial algorithm for ring networks.

References

[1] A. D. Birrel, R. Levin, R. M. Needham, and M. D. Schroeder, "Grapevine: an exercise in distributed computing," *Communications of the ACM*, vol. 25, pp. 260–274, April 1982.

[2] D. Terry, "Caching hints in distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 48–54, January 1987.

[3] S. J. Mullender and P. M. Vitány, "Distributed match-making," *Algorithmica*, no. 3, pp. 367–391, 1988.

[4] R. Fowler, "The complexity of using forwarding addresses for decentralized object finding," in *Fifth ACM Symposium on the Principles of Distributed Computing*, (Calgary, Alberta, Canada), pp. 108–120, ACM, August 11–13 1986.

[5] R. Fowler, "Decentralized object finding using forwarding addresses," PhD Thesis 85-12-1, University of Washington, December 1985.

[6] B. O. Koopman, *Search and Screening.* Pergamon Press, 1980.

[7] L. D. Stone, *Theory of Optimal Search.* Vol. 118 of *Mathematics in Science and Engineering*, Academic Press, 1975.

[8] L. D. Stone and J. A. Stanshine, "Optimal search using uninterrupted contact investigation," *SIAM Journal of Applied Mathematics*, vol. 20, pp. 241–263, March 1971.

[9] Y. Kan, "Optimal Search of a Moving Target," *Operations Research*, vol. 25, pp. 864–870, September–October 1977.

[10] N. Megiddo, S. Hakimi, M. Garey, D. Johnson, and C. Papadimitriu, "The complexity of searching a graph," *J. ACM*, vol. 35, pp. 18–44, Jan. 1988.

[11] T. D. Parsons, "Pursuit–evasion in a graph," in *Theory and Applications of Graphs*, (Y. Alavi and D. Lick, eds.), pp. 426–441, Berlin: Springer–Verlag, 1976.

[12] T. D. Parsons, "The search number of a connected graph," in *Proceedings of the* 9^{th} *South Eastern Conference on Combinatorics, Graph Theory and Computing*, (Winnipeg, Canada), pp. 549–554, Utilitas Mathematica, 1978.

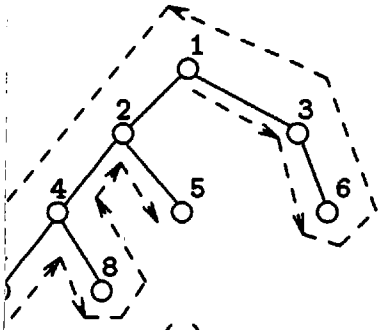
[13] M. Ahamad, M. Ammar, J. Bernabéu–Aubán, and Y. Khalidi, "Using Multicast Communication to Locate Resources in a LAN-Based Distributed System," in *Proceedings of the 13th Conference on Local Computer Networks*, IEEE, October 1988.

[14] J. Bernabeu, M. Ammar, and M. Ahamad, "Optimal selection of multicast groups for resource location in a distributed system," in *Proceedings of IEEE INFOCOM*, IEEE, 1989.

[15] K. Trummel and J. Weisinger, "The complexity of the optimal searcher path problem," *Operations Research*, vol. 34, pp. 324–327, March–April 1986.

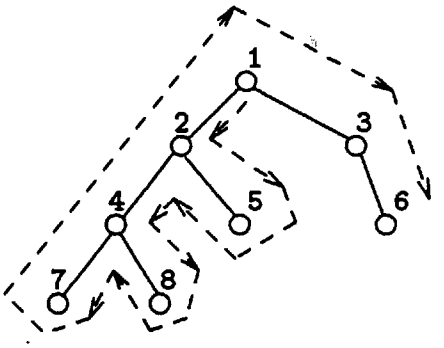
[16] N. M. Maxemchuck, "Routing in the manhattan street network," *IEEE Transactions on Communications*, vol. 35, pp. 503–512, May 1987.

[17] F. Borgonovo and E. Cadorin, "Routing in the bidirectional manhattan network," in *Data Communications Conference*, (Rio de Janeiro), 1987.



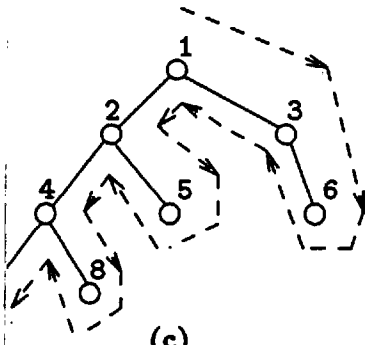
(a)

Uniform Π
$c(e) = 1$ for all $e \in E$.



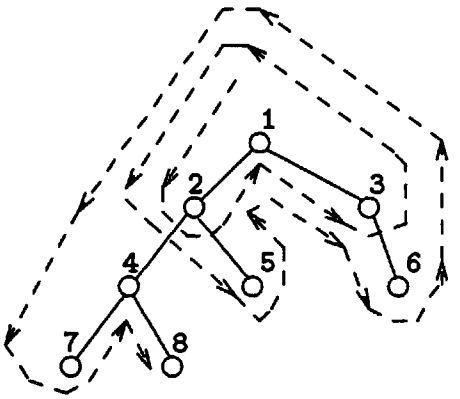
(b)

$\Pi(v) = 0.01$, for $v \in \{2, 3, 4\}$
$\Pi(v) = 0.2425$ for $v \in \{5, 6, 7, 8\}$
$c(e) = 1$ for all $e \in E$.



(c)

$) = 0.01$, for $v \in \{2, 3, 4\}$
$) = 0.2425$ for $v \in \{5, 6, 7, 8\}$
$4, 7\}) = c(\{4, 8\}) = 2$
$) = 1$ for all other $e \in E$



(d)

$\underline{\Pi} = (0.5, 0.4, 0.002, 0.08, 0.014, 0.002, 0.002)$
$c(e) = 1$ for all $e \in E$

Figure 3: Example of optimal traversals in tree networks

To appear in ACM Transactions on Computer Systems.

Multi-Dimensional Voting*

Mustaque Ahamad†
Mostafa H. Ammar†
Shun Yan Cheung‡

†College of Computing
Georgia Institute of Technology, Atlanta, GA 30332

‡Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322

Abstract

We introduce a new concept, *multi-dimensional voting*, in which the vote and quorum assignments are k-dimensional vectors of non-negative integers and each dimension is independent of the others. Multi-dimensional voting is more powerful than traditional weighted voting because it is equivalent to the general method for achieving synchronization in distributed systems which is based on sets of groups of nodes (quorum sets). We describe an efficient algorithm for finding a multi-dimensional vote assignment for any given quorum set and show examples of its use. We demonstrate the versatility of multi-dimensional voting by using it to implement mutual exclusion in fault-tolerant distributed systems, and protocols for synchronizing access to fully and partially replicated data. These protocols cannot be implemented by traditional weighted voting. Also, the protocols based on multi-dimensional voting are easier to implement and/or provide greater flexibility than existing protocols for the same purpose. Finally, we present a generalization of the multi-dimensional voting scheme, called *nested multi-dimensional voting*, that can facilitate implementation of replica control protocols that use structured quorum sets.

*This work was supported in part by NSF grants NCR-8604850 and CCR-8806358, and by the University Research Committee of Emory University.

1 Introduction

Distributed systems offer many advantages, including resource sharing and fault-tolerance. The latter can be achieved by replicating a resource at nodes with independent failure modes. Replication can also improve performance when load is shared among the nodes that have instances of a resource. In many applications, users need to synchronize access to shared resources. For example, when data is replicated to improve its availability, updating the data requires mutually exclusive access. This is necessary for maintaining data consistency. The synchronization technique should work in the presence of node and communication failures.

Quorum consensus is a general class of synchronization protocols for distributed systems. An operation proceeds to completion only if it can obtain permission from nodes that constitute a *quorum group* [3]. Quorum groups used by conflicting operations have non-empty intersections to guarantee proper synchronization. The collection of quorum groups used by an operation is known as a *quorum set*. When each group intersects with every other group in a quorum set, it is called a *coterie* [4, 5] and it can be used to achieve mutual exclusion in distributed systems. The general method to define quorum sets is by listing them explicitly. A well-known method for defining quorum sets is weighted voting [6] which is a generalization of the majority consensus method [7]. In voting, each node is assigned a number of votes and each operation must obtain a pre-defined quorum of votes before it is allowed to execute to completion. Voting can be used for achieving mutual exclusion and synchronizing reading and writing of replicated data. In mutual exclusion, each operation must obtain a majority of the votes assigned before it can proceed. In reading and writing, the read and write quorums must be such that their sum is more than the total number of votes and when version numbers are used to identify the most recent update, the write quorum should be at least a majority of all votes.

Implementing quorum sets in general requires that each node maintain a list of member groups. A decision on whether a quorum has been collected is achieved through a search of that list. In general, to determine if a group of nodes that responded to a request is a quorum group is time consuming because the size of the quorum set can be exponential. Weighted voting, on the other hand, is easier to implement as each node has to maintain its own vote assignment. An operation can proceed if the number of votes collected is at least

the required quorum. Also, addition and removal of nodes may cause the quorum sets to be purged and replaced by new ones but will only require a change in the quorum assignment when voting is used.

It was shown in [5] that the method of quorum sets is more general than voting by showing quorum sets which cannot be obtained from any vote assignment. Quorum sets that are not obtained from vote assignments can be used to achieve better performance by reducing the number of messages. The quorum consensus methods proposed in [8], [9], [10] and [11] organize the nodes in a logical structure. The quorum groups that are derived from such structures can be smaller than those defined by voting and have lower communication cost. These *structured* quorum sets are usually not defined by vote assignments.

We present in this work a new unifying voting-based method that is as powerful as the method of quorum sets and has the flexibility and ease of implementation of voting. In *multi-dimensional* (MD) voting, the vote assignment to each node and the quorums are k-dimensional vectors of non-negative integers. Each dimension of the vote and quorum assignment is similar to voting and the quorum requirements in different dimensions can be combined in a number of ways. This makes multi-dimensional voting more powerful than standard voting. We will discuss a number of applications which can be implemented with multi-dimensional voting but not with standard voting.

Methods based on coteries or voting when the quorum or vote assignment does not change are static because the groups of nodes that can allow an operation to complete do not change. Thus, systems that use a static method do not attempt to adapt to continuously varying system state. Dynamic protocols in contrast react to changes in the state of the system and adapt the synchronization procedure accordingly. The system can use the state information to determine the best quorum set or to reconfigure itself in anticipation of future failures. Several dynamic quorum consensus methods have been proposed (see for example [12], [13], [14] and [15]). These protocols can use different quorum sets at different times and the various pairs of read and write quorum sets used must satisfy additional constraints to guarantee data consistency. In this paper we focus on the description of MD-voting and its use in static quorum consensus protocols. A dynamic MD-voting method similar to the quorum inflation/deflation technique in [13] has been presented in [16].

The paper is organized as follows. In Section 2, we introduce the concept of a multi-

dimensional vote assignment. In Section 3, we show that every quorum set can be represented by a multi-dimensional vote assignment and present an efficient algorithm for finding one. Sections 4 and 5 discuss the use of multi-dimensional voting for mutual exclusion and reading and writing of replicated data, respectively. In Section 6 we consider the use of MD-voting for maintaining the consistency of partially replicated data where individual nodes may not contain the entire replica. Such a scheme is described in [17]. By applying the MD-voting paradigm to the problem, we develop a more flexible scheme that provides higher availability in some instances. Finally, we present in Section 7, a generalization called *Nested MD-voting* and provide an example of its use. We conclude the paper in Section 8.

2 Multi-Dimensional Voting

We consider a distributed system of N nodes which are numbered as 1, 2, ..., N. In multi-dimensional (MD) voting, the vote value assigned to a node and the quorum are k-dimensional vectors of non-negative integers. Formally, the MD vote assignment $V_{N,k}$ is a Nxk matrix where $v_{i,j}$ represents the vote assignment to node i in the j^{th} dimension and $v_{i,j} \geq 0$ for $i = 1,2,\ldots,N$ and $j = 1,2,\ldots,k$. The votes assigned in the various dimensions are independent of each other. The quorum assignment $\underline{q_k} = (q_1, q_2, \ldots, q_k)$ is a k-dimensional integer vector, where $q_j > 0$, for $j = 1,2,\ldots,k$. In addition, a number ℓ, $1 \leq \ell \leq k$, is defined which is the number of dimensions of vote assignments for which the quorum must be satisfied. Thus, there are *two* levels of requirements: vote and dimension. At the vote level, the number of votes received for a dimension must be greater than or equal to the quorum requirement in that dimension. At the dimension level, the number of dimensions for which a quorum is collected must be greater than or equal to ℓ. As we show in the next section, this extra level of flexibility makes MD-voting more powerful than standard voting. We denote MD-voting with quorum requirement in ℓ of k dimensions as MD(ℓ, k)-voting and the term SD-voting (single dimensional voting) will refer to the standard voting method described in [6]. In fact, MD(1,1)-voting is the same as SD-voting.

Synchronization methods developed from MD-voting operate in a similar manner as SD-voting. Each node stores its vote which consists of k integers and each operation has a quorum requirement for each dimension and the value of ℓ. An operation requests permission from the nodes by sending a voting request to them. When a node receives a vote request, it

votes "reject" if it wants to disallow the operation to proceed (e.g., due to locking conflict) or replies with its vote in all dimensions. Each operation maintains k independent variables which accumulate the votes received in each dimension. When a response containing a vote is received, the operation adds the vote in each dimension to the appropriate variable and when the sums in at least ℓ variables are greater than or equal to the quorum in the corresponding dimensions, the operation can proceed.

Figure 1 shows an example of how MD-voting works in a system of six nodes. Let us assume that in response to a request, the votes of nodes 1, 3 and 4 are received. These vectors are added up and the vector sum is then compared with the vector quorum assignment. The compare operation is performed per dimension and the results zero and one represent quorum deficiency and sufficiency, respectively. In this case, the quorum requirement is satisfied only in the fourth dimension. If MD(1,4)-voting is used, then an operation collecting the votes shown in Figure 1 will be allowed to proceed. However, the operation cannot be executed when MD(ℓ, 4)-voting, for ℓ = 2, 3, and 4, is used.

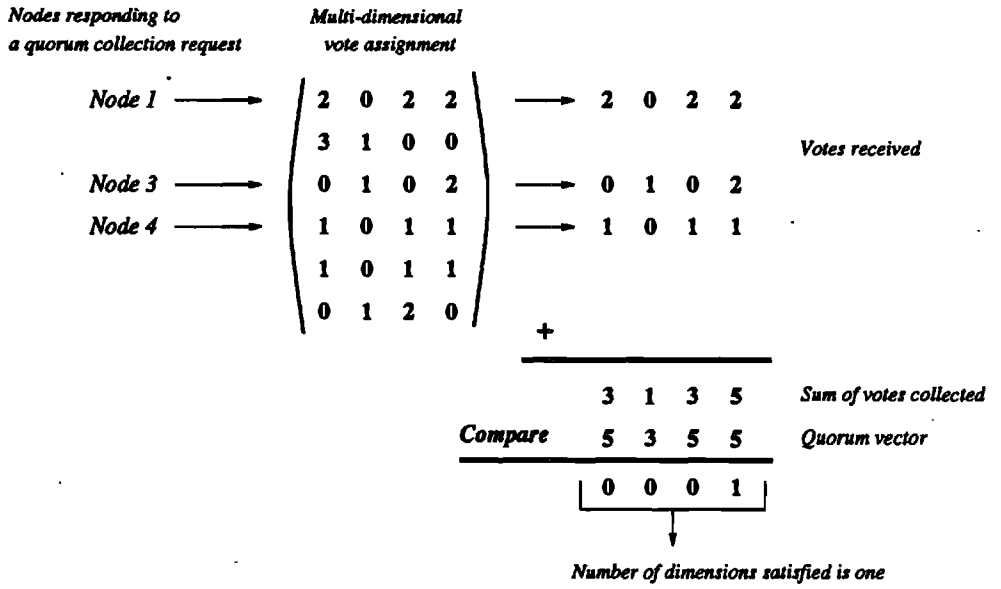


Figure 1: Voting procedure in multi-dimensional voting

One way to implement MD-voting is for each node to store the k integers representing its vote and quorum assignment. In response to a voting request, a node sends the integers representing the vote assignment of the sending node. An alternative implementation is to

have all nodes store the quorum vector and the vote assignment matrix. Voting messages in this case contain only the identifier of the sending node; thus trading off storage with message length. When a large number of dimensions is used, the voting messages can be long or significant storage space may be required. In the next section, we will present a method for finding MD vote and quorum assignments which tries to reduce the number of dimensions.

3 Finding a Multi-Dimensional Vote Assignment

3.1 Definitions and Notation

Let $U = \{1, 2, \ldots, N\}$ be the universe set of all nodes and we will refer to sets of nodes as *groups*. A quorum set Q is a set of groups of nodes in U and these groups have the *minimality* property [3]:

$$\forall\, G, H \in Q: \;\; G \not\subseteq H$$

The synchronization requirements define what groups are included in the set. For example, if mutual exclusion is desired, this set is a coterie and any two of its members must have a non-empty intersection (see Section 4).

A number of quorum sets can be represented by SD-voting. Each node i in SD-voting is assigned v_i votes ($1 \leq i \leq N$) where v_i is a non-negative integer and a quorum q is defined, such that nodes in each group of the set have at least q votes. Specifically, with the vote assignment $\underline{v} = (v_1, v_2, \ldots, v_N)$, the members of the set defined by $(\underline{v}, q)$ are *tight* groups of nodes which have at least q votes. A group G is tight with respect to quorum q if,

- $\sum_{g \in G} v_g \geq q$ and,
- any proper subset of G has less than q votes

The set of tight groups Q defined by $(\underline{v}, q)$ is,

$$Q = \{G \mid G \text{ is a tight group with respect to quorum } q\}$$

and this set has the minimality property since if there would exist $G, H \in Q$ such that $G \subset H$, then H would not be tight. Hence, a vote and a quorum assignment $(\underline{v}, q)$ defines

a *unique* quorum set. For instance, the vote assignment (1,1,1) to a three node system with $q = 2$ defines the quorum set {{1,2},{1,3},{2,3}}. The same set can be represented by the vote assignment (2,2,3) and $q = 4$. In fact, if a quorum set is SD-vote assignable, there is an infinite number of vote and quorum assignments that may be used to represent it.

In a similar manner, an MD(ℓ, k) vote and quorum assignment also defines a unique quorum set. A group G is a *tight* group in MD(ℓ, k)-voting with respect to quorum requirement $\underline{q}$ if

- $\sum_{g \in G} v_{g,j_i} \geq q_{j_i}$ for at least ℓ distinct dimensions $j_1, j_2, \ldots, j_\ell$ and,
- any proper subset of G satisfies quorum requirements in strictly less than ℓ dimensions

The set $Q_{\ell,k}(V_{N,k}, \underline{q}_k)$ of tight groups represented by the MD(ℓ, k) vote and quorum assignment $(V_{N,k}, \underline{q}_k)$ is,

$$Q_{\ell,k}(V_{N,k}, \underline{q}_k) = \{G \mid G \text{ is a tight group in MD}(\ell, k)\text{-voting with respect to } \underline{q}_k\}$$

Similar to SD-voting, the same set of tight groups can be represented by different MD(ℓ, k) vote and quorum assignments. The set of tight groups for the special cases where $\ell = 1$ (any dimension) and $\ell = k$ (all dimensions) can be given as follows,

$$Q_{1,k}(V_{N,k}, \underline{q}_k) = \{G \mid G \text{ is a tight group such that: } \exists j : 1 \leq j \leq k : \sum_{g \in G} v_{g,j} \geq q_j\}$$

$$Q_{k,k}(V_{N,k}, \underline{q}_k) = \{G \mid G \text{ is a tight group such that: } \forall j : 1 \leq j \leq k : \sum_{g \in G} v_{g,j} \geq q_j\}$$

In MD(1,k)-voting, an operation can proceed if quorum is available in any dimension and in MD(k, k)-voting, quorum requirements in all dimensions must be satisfied. For MD(1,k)-voting, we can also write,

$$Q_{1,k}(V_{N,k}, \underline{q}_k) = \{G \mid G \in \bigcup_{j=1}^{k} C_j \wedge \forall H \in \bigcup_{j=1}^{k} C_j : H \not\subset G\} \tag{1}$$

where C_j is the set of tight groups defined by the j^{th} dimension of vote and quorum assignment, i.e, $Q_{1,k}(V_{N,k}, \underline{q}_k)$ is all the minimal groups in $\bigcup_{j=1}^{k} C_j$.

Table 1 presents a two-dimensional vote and quorum assignment to a system of four nodes. The sets C_1 and C_2 are the sets of tight groups corresponding to the first and second dimension of the MD vote and quorum assignment, respectively.

$$V_{4,2} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 0 & 2 \end{pmatrix}, \qquad \underline{q}_2 = (2,3)$$

$C_1 = \{\{1,2\},\{1,3\},\{2,3\}\}, C_2 = \{\{1,4\},\{2,4\},\{3,4\},\{1,2,3\}\}$

$Q_{1,2}(V_{4,2}, \underline{q}_2) = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\}$

$Q_{2,2}(V_{4,2}, \underline{q}_2) = \{\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\}\}$

Table 1: An example of multi-dimensional vote assignment

3.2 The Existence of MD Vote Assignments

We show that any quorum set Q can be represented by an MD(1,k) vote and quorum assignment.

Lemma 3.1: Let Q be a set of groups satisfying the minimality property such that

$$\forall\, G, H \in Q: \;\; G \not\subseteq H$$

Then Q can be represented by an MD(1,k) vote and quorum assignment where $k = |Q|$.

Proof: Let $Q = \{G_1, G_2, \ldots, G_k\}$ so that $|Q| = k$. We construct the following k-dimensional vote assignment: the vote value of node i in the j^{th} dimension, $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, k$ is given by,

$$v_{i,j} = 1 \qquad \text{for } i \in G_j$$
$$v_{i,j} = 0 \qquad \text{for } i \notin G_j$$

with $q_j = |G_j|$. We will show that $Q = Q_{1,k}(V_{N,k}, \underline{q}_k)$.

From the construction of the MD(1,k) vote and quorum assignment that yields $Q_{1,k}(V_{N,k}, \underline{q}_k)$, it is trivially true that $H \in Q \implies H \in Q_{1,k}(V_{N,k}, \underline{q}_k)$, i.e., $Q \subseteq Q_{1,k}(V_{N,k}, \underline{q}_k)$. (When $H = G_j$, votes from nodes in H satisfy the quorum requirement in the j^{th} dimension and H is also tight.) Consider the j^{th} dimension of the MD vote assignment that is derived from the group $G_j \in Q$. The set of groups C_j represented by this dimension is equal

to $\{G_j\}$ and since $Q_{1,k}(V_{N,k}, \underline{q}_k)$ is all the *minimal* groups in $\bigcup_{j=1}^{|Q|} C_j$ (see (1)), we have $Q_{1,k}(V_{N,k}, \underline{q}_k) \subseteq Q$.□

Lemma 3.1 guarantees that an MD(1,k) vote and quorum assignment can be found for any set of minimal groups Q. In the constructive proof, since each group is represented by a separate dimension, the number of dimensions used is equal to $|Q|$, which may be large. We present in what follows a technique that tries to represent several groups by a single dimension of an MD vote and quorum assignment. Therefore, in practice, the number of dimensions needed to obtain an MD(1,k) vote assignment could be much less than $|Q|$.

3.3 Algorithm for Finding an MD Vote Assignment

The proof of Lemma 3.1 provides an MD(1,k) vote and quorum assignment with k equal to the number of groups in the quorum set. In this assignment, each dimension represents one quorum group and vice versa. Since all groups of an SD-vote assignable quorum set can be represented by a single dimension, MD(1,1) voting can be used to represent such sets. For quorum sets that are not SD-vote assignable, we use a single dimension to represent as many groups as possible to reduce the number of dimensions.

In [18], a technique is described for testing if a set of groups Q is SD-vote assignable. A linear program, LP(Q), is set up using the groups in Q and solved using the Simplex method. If LP(Q) does not have a feasible solution then Q is not SD-vote assignable. Otherwise, a rational solution is found which can be converted to an integral vote and quorum assignment. We have extended the algorithm to find an MD(1,k) vote and quorum assignment for a quorum set Q. The new algorithm is illustrated in Figure 2 and is described in detail in Appendix A. It finds an MD(1,k) vote assignment by testing to see if the initial quorum set Q is SD-vote assignable. If so, the algorithm outputs the MD(1,1)-vote and quorum assignment found by the Simplex method. Otherwise, a group A is removed from Q and the quorum set $Q - \{A\}$, consisting of the remaining groups of Q, is tested to see if it is SD-vote assignable. This process is repeated until the groups that remain form an SD-vote assignable quorum set. The vote and quorum assignment for this quorum set is the first dimension of the MD-vote and quorum assignment. The groups that were removed in the process are stored in the temporary variable D and used as input in a second iteration to find the second dimension of vote and quorum assignment. This is repeated until all groups

are represented by the MD-vote and quorum assignments. Since a quorum set consisting of a single group is SD-vote assignable, at least one group of Q is represented and removed in each iteration, and the algorithm is guaranteed to terminate.

This algorithm will be used to find MD(1,k) vote and quorum assignments for several non-SD vote assignable quorum sets in the following sections.

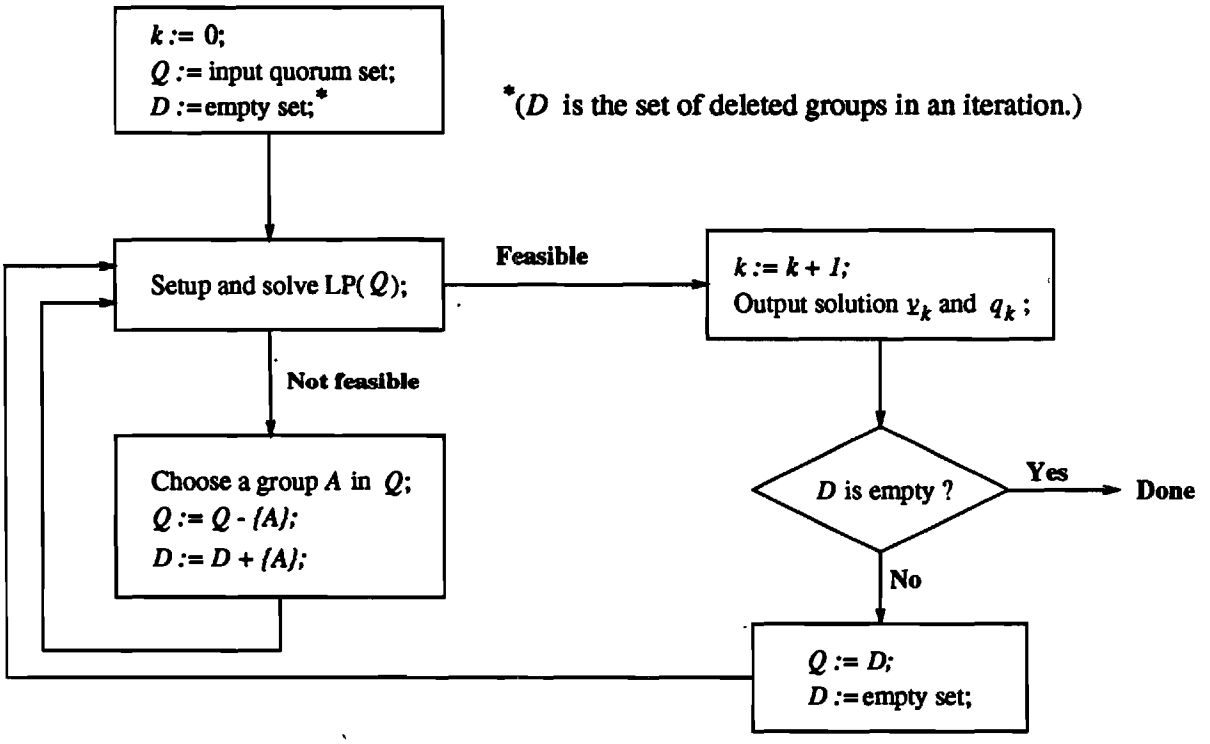


Figure 2: Algorithm for finding MD(1,k) vote and quorum assignment

3.4 Using MD-Voting for Distributed Synchronization

MD-voting is a powerful concept and has a wide range of applications. The subsequent sections (Sections 4, 5 and 6) illustrate the versatility of MD-voting by using it to implement solutions to a variety of synchronization problems in distributed systems. Although any quorum set can be represented by an MD(1,k) vote and quorum assignment, finding such a representation, however, requires listing the quorum set and using the algorithm in the previous subsection. For certain replica control protocols producing the listing is imprac-

tical. In such situations, we would like to be able to determine the MD-voting parameters *from the description* of the protocol. In some instances, as will be seen in Section 5.2, MD(ℓ, k)-voting parameters may be found from the protocol description. In other situations, this may not be possible. We introduce a generalization of MD-voting, called Nested MD-voting, which will help us in determining the MD-voting parameters for a hierarchical quorum consensus method.

4 MD-Voting for Mutual Exclusion

The problem of mutual exclusion arises in many applications where a process must acquire exclusive access to a shared resource. In distributed systems, the synchronization method used must tolerate node and link failures. The general method for achieving synchronization in a distributed system is the use of *coteries*. The definition of a coterie is given in [5] and it is repeated here for completeness.

Definition 4.1: *Coterie* [5]. A set of groups Q is a *coterie* under $U = \{1, 2, \ldots, N\}$ iff

1. $G \in Q \implies G \subseteq U \ \wedge \ G \neq \phi$
2. (Intersection property) $\forall\, G, H \in Q : \ G \cap H \neq \phi$
3. (Minimality property) $\forall\, G, H \in Q : \ G \not\subseteq H$

A process synchronizes with other processes by obtaining permission from nodes that form a group of the coterie. A node gives permission to only one request at a time and the other requests are kept pending until the request that was given permission completes. The intersection property guarantees that only one process will succeed at a time and mutual exclusion is achieved. However, in the general case, implementation of the method based on coteries could be complex. It will require that processes keep a list of the groups in the coterie and a comparison of the responses against this list is made to determine if a process can proceed. To determine if a set of nodes form a group in a coterie can be computationally expensive because the size of a coterie can be exponential.

SD-voting can also be used to achieve mutual exclusion when the quorum used is a majority of the votes. The SD vote assignment to the nodes uniquely determines a coterie

and we will call coteries that have an SD-voting equivalent *SD-vote assignable*. There exist coteries that cannot be obtained from SD vote assignments, thus the method of coteries is more general than SD-voting [5]. SD-voting also requires a relatively large number of nodes to participate in the execution of the protocol. For example, to achieve mutual exclusion, nodes that have more than half the votes must participate. Consequently, each mutual exclusion request generates a large number of messages which have a significant impact on response time. Coteries that are not SD-vote assignable can achieve mutual exclusion using a lower number of messages (e.g. [8] and [9]). However, such coteries cannot be implemented by SD-voting. The following corollary shows that MD-voting is as general as coteries and can be used to implement mutual exclusion methods with the desirable properties of SD-voting.

Corollary 4.1: A mutual exclusion coterie Q of N nodes can be represented by an MD(1,k) vote and quorum assignment.

Proof: Since Q satisfies the minimality property (Q is a quorum set), the claim follows from Lemma 3.1.

Table 2 shows a coterie which was described in [5] and it was shown to be non SD-vote assignable. The table presents an MD(1,4) vote and quorum assignment for the coterie. (Notice that the number of dimensions, 4, is smaller than the number of groups in the coterie which is 7.)

$Q = \{\{12\},\{134\},\{135\},\{146\},\{156\},\{236\},\{245\}\}$
$V_{6,4} = \begin{pmatrix} 2 & 0 & 2 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}, \quad \underline{q}_4 = (5,3,5,5)$

Table 2: A non SD-vote assignable coterie and its multi-dimensional vote assignment

A structured coterie approach organizes nodes in some logical structure and groups in the coterie are derived from this structure. In [9] the nodes are organized into a binary tree structure and groups of nodes that form a path from the root to a leaf define a group in the coterie. If some node on a path fails, it is replaced by two paths starting from the children of the failed node. The tree-based method can achieve mutual exclusion using as few as $\log(N)$ messages when there are few failures. Table 3 shows the MD(1,5)-vote and quorum assignment, produced by the algorithm in Section 3, for the coterie that is derived from a binary tree of depth three. Alternatively, we can find, by inspection, the MD(2,3)-vote and quorum assignment in Table 4 to represent the same coterie.

Q = {{124},{125},{136},{137},{145},{167},{2346},{2347}, {2356},{2357},{2467},{2567},{3456},{3457},{4567}}

$$V_{7,5} = \begin{pmatrix} 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 2 & 0 & 2 & 0 & 1 \\ 2 & 2 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}, \quad \underline{q}_5 = (6,6,6,4,4)$$

Table 3: A tree-based coterie and its MD(1,5)-vote and quorum assignment

$$V_{7,3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 2 \end{pmatrix}, \quad \underline{q}_5 = (1,3,3)$$

Table 4: MD(2,3)-vote and quorum assignment for the tree-based coterie inTable 3

5 MD-Voting for Reading and Writing Replicated Data

We examine the use of MD-voting to implement read and write quorum sets which correspond to replica control protocols that cannot be implemented using SD-voting. We assume that a node stores only a single version of the replica and version numbers are used to identify the most recently updated replicas. The method described can easily be modified when timestamping is used instead of version numbers. A read operation returns some value and a write operation installs a new value. To ensure *one-copy serializability* [19], we do not allow a read and write operation and two write operations to execute concurrently[1]. In general, synchronization of read and write operations to replicated data can be ensured by requiring that each operation obtain permission of a group of nodes and the groups used by conflicting operations have non-empty intersection. Minimal groups of nodes that can allow a read and write operation to complete are called read and write groups respectively. The read and write quorum sets R and W are the sets of read and write groups used. (W is a quorum set and R is its anti-quorum set [3].) The synchronization requirements given above are satisfied if,

1. (Read/write intersection property) $\forall\, G \in R, H \in W : G \cap H \neq \phi$, and
2. (Write/write intersection property) $\forall\, G, H \in W : G \cap H \neq \phi$.

R and W have the minimality property and W also has the intersection property (W is a coterie). Since read operations can be executed concurrently, the set R need not satisfy the intersection property, i.e., in general R is not a coterie. For a given R, to maximize write availability, W equals the *maximal* set of minimal groups that have read/write and write/write intersection property. The set W, in general, is not unique for a given R. For example, let $R = \{\{1,2\},\{3,4\}\}$, then the sets {{1,3},{1,4},{2,3,4}} and {{1,3},{2,3},{1,2,4}} can both be used as write quorum sets.

A replica control protocol corresponds to a read and a write quorum set which satisfy the synchronization requirements. Thus, in the general case, consistency of replicated data

[1]In addition to the replica control protocol each node uses a local synchronization protocol (e.g., two-phase locking or timestamp ordering) to achieve one-copy serializability. Note that single writer/multiple readers synchronization is sufficient but not necessary for achieving one-copy serializability.

is maintained using two possibly different sets of groups (one for reading and the other for writing) which have the minimality property. It is straightforward to see that MD vote assignments can be obtained for each of them, one is used for reading and the other one is used for writing. In the general case, the MD vote assignments obtained for the read and write quorum sets may be different. Consequently, a node must use the appropriate MD vote assignment (based on the type of the request) to vote on each request. Thus, votes obtained for a read request cannot be used for a write request. Although it is feasible to implement read and write quorum sets with separate MD vote and quorum assignments, it is simpler and more efficient to allow votes obtained for reading to be augmented to a quorum for writing because transactions usually read the data before updating it. This will be similar to SD-voting where the *same* vote assignment is used to define both read and write quorum sets. In the next subsection, we describe a replica control protocol that uses a single MD vote assignment to define both read and write quorum sets and allows a read quorum to be augmented when the read data items are also updated.

5.1 A Replica Control Protocol Based on MD-Voting

In the design of a replica control protocol under the assumption that read transactions are predominant, an appropriate read quorum set that provides high read performance is chosen and the corresponding write quorum set is computed to satisfy the synchronization requirements. In general, the read quorum set can be represented by an MD(ℓ, k) vote and read quorum assignment. Let $V_{N,k}$ and $\underline{r}_k = (r_1, r_2, \ldots, r_k)$ be the vote and read quorum assignment for an MD(ℓ, k)-voting system used in reading and $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ represents the set of minimal groups defined by the assignment. To allow write operations to synchronize using the same vote and read quorum assignment, we define the write quorum $\underline{w}_k = (w_1, w_2, \ldots, w_k)$ to be,

$$w_j = \sum_{i=1}^{N} v_{i,j} - r_j + 1, \quad \text{for } j = 1, 2, \ldots, k$$

Since, as explained later, the write/write intersection is achieved in another manner, we *do not* require that $w_j \geq \left\lceil \frac{\sum_{i=1}^{N} v_{i,j}+1}{2} \right\rceil$ votes, for $j = 1, 2, \ldots, k$. The write quorum w_j will only ensure that groups that satisfy the write requirement intersect with all read groups of the j^{th} dimension of the MD vote assignment. Since the read quorum set is defined

by MD(ℓ, k)-voting, we must use MD$(k - \ell + 1, k)$-voting for writing to ensure that the read/write intersection property holds. Let $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$ be the set of tight groups represented by the MD$(k + 1 - \ell, k)$ vote and *write* quorum assignment. The following lemma shows that $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ and $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$ have the read/write intersection property.

Lemma 5.1:

$$\forall\, G \in Q_{\ell,k}(V_{N,k}, \underline{r}_k),\ H \in Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k) :\ G \cap H \neq \phi$$

Proof:

Let G and H be two arbitrary groups in $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ and $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$ respectively. Since $(\ell) + (k + 1 - \ell) > k$, there is some dimension s such that,

$$\sum_{g \in G} v_{g,s} \geq r_s$$

$$\text{and} \quad \sum_{h \in H} v_{h,s} \geq w_s$$

Since $r_s + w_s > \sum_{i=1}^{N} v_{i,s}$, there must be a common node in G and H and hence $G \cap H \neq \phi$.□

Although the sets $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ and $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$ have the intersection property which is necessary for read/write synchronization, $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$ may not be a write quorum set for $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ because it may not have the write/write intersection property which is required when version numbers are used. To achieve this, we can augment each group of $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$ to include a group of $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$. We define the write quorum set W which is derived from $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ and $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$ in the following way:

$$W = \{A \cup B \mid A \cup B \text{ is minimal and } A \in Q_{\ell,k}(V_{N,k}, \underline{r}_k), B \in Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)\}$$

The set W is unique for a given pair $(Q_{\ell,k}(V_{N,k}, \underline{r}_k), Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k))$ and it can be constructed by first forming the set of all groups $\{A \cup B\}$, where $A \in Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ and $B \in Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$, and then removing all the groups that are supersets of some other group in the constructed set. It can be easily seen that when $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ and W are

used for reading and writing, both the read/write and write/write intersection properties are satisfied. The latter property follows from the read/write intersection of groups in $Q_{\ell,k}(V_{N,k}, \underline{r}_k)$ and $Q_{k+1-\ell,k}(V_{N,k}, \underline{w}_k)$.

The replica control protocol used is thus as follows: when reading, the operation obtains a read quorum in at least ℓ dimensions and when writing, it must obtain a read quorum *and* a write quorum in ℓ and $k+1-\ell$ dimensions, respectively. If reading of the data is followed by its write (which is the typical case), the write operation only needs to obtain a write quorum and the method thus allows the read quorum to be augmented.

A special case of the protocol is when MD(1,k)-voting is used for reading. Then, we can use the method in Section 3 to find an MD(1,k) vote and read quorum assignment for any given read quorum set. The corresponding write quorum set will be represented by using an MD(k,k) vote and write quorum assignment. In this case, the read operation can proceed if it can obtain a read quorum in any one dimension. If the transaction wishes to update the data after reading it, the votes received for the read request must be supplemented with additional votes such that in each dimension the number of votes received is greater than or equal to the write quorum for that dimension. The general case where MD(ℓ,k)-voting (arbitrary ℓ) is used, is more difficult as we do not yet have an algorithm to find an MD(ℓ,k) assignment when $\ell \neq 1$. However, if the read quorum set used is derived from some logical structure, such as the example described in the next subsection, we may be able to use the structure to formulate an MD assignment.

5.2 Example - The Grid Protocol

As an example, we consider the replica control method presented in [10] which organizes the nodes of the system into a logical grid consisting of m rows and n columns. A read quorum group contains n nodes where one node is selected from each column and a write group consists of nodes in a read group and all nodes in a column of the grid (the quorum sets used in this method are generally not SD-vote assignable). For example, Figure 3 shows a six node system organized into a 2x3 grid. The read and write quorum sets used are {{1,2,3}, {1,2,6}, {1,5,3}, {1,5,6}, {4,2,3}, {4,2,6}, {4,5,3}, {4,5,6}} and {{1,4,2,3}, {1,4,2,6}, {1,4,5,3}, {1,4,5,6}, {2,5,1,3}, {2,5,1,6}, {2,5,4,3}, {2,5,4,6}, {3,6,1,2}, {3,6,1,5}, {3,6,4,2}, {3,6,4,5}}, respectively. Notice that a read and a write quorum group and two

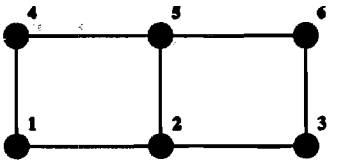


Figure 3: A Grid Network

write quorum groups have non-empty intersection.

The read quorum groups of the grid protocol can be represented by the following MD-vote and quorum assignment. The MD-vote assignment used for an mxn grid network consists of n dimensions and a node i has $v_{i,j} = 1$ if it is in column j, otherwise $v_{i,j} = 0$, for $j = 1, 2, \ldots, n$. For instance, the MD-vote assignment of the 2x3 grid system in Figure 3 is given in Table 5. Nodes in the first column (i.e., nodes 1 and 4) are assigned one vote in the

$$V_{6,3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \underline{r}_3 = (1,1,1), \quad \underline{w}_3 = (2,2,2)$$

Table 5: Multi-dimensional vote and quorum assignment for the 2x3 grid system

first dimension and zero votes in the other dimensions. Similarly, the nodes in the second and third columns are assigned one vote in the second and third dimensions, and zero votes in the other dimensions, respectively. The read quorum vector used is $\underline{r}_n = (1, 1, ..., 1)$ and the voting method is MD(n, n)-voting (i.e., quorum requirement must be satisfied in all n dimensions). The number of votes collected in dimension j is at least one if and only if some node in column j responded positively to a voting request. If the quorum requirement is satisfied in all dimensions, then there is at least one node in each column that responded positively, and vice versa. Thus, the MD(n, n)-voting method represents the read quorum set of the grid protocol.

There are many write quorum sets possible for a given read quorum set of the grid

protocol and the one proposed by the previous subsection is defined using the write quorum vector $\underline{w}_n = (w_1, w_2, \ldots, w_n)$ where $w_j = \sum_{i=1}^{mn} v_{i,j} - r_j + 1$, for $j = 1, 2, \ldots, n$. Since $r_j = 1$, for all j, and only nodes in column j have non-zero votes in the j^{th} dimension, we have that $w_j = m$, for all j. A write quorum group consists of the union of a read quorum group and a group in the set $Q_{1,n}(V_{N,n}, \underline{w}_n)$ which is the set of tight groups defined by MD(1,n)-voting. A group in $Q_{1,n}(V_{N,n}, \underline{w}_n)$ satisfies the quorum requirement in at least one dimension. Since $w_j = m$ and only nodes in column j have non-zero votes in dimension j, for $j = 1, 2, \ldots, n$, the quorum requirement in dimension j is satisfied only when all nodes in column j respond positively. Hence, a write quorum group consists of a read quorum group and all nodes in a column of the grid which is the case in the grid protocol.

The grid protocol is based on the structured quorum set concept discussed in Section 4. The size of the read and write groups in a square grid is of the order $O(\sqrt{K})$, where K is the number of nodes with replicas and it can be smaller than quorum groups in SD-voting. A simulation study in [10] showed that the response times of transactions in systems using the grid protocol are significantly lower than those that use SD-voting for the same number of nodes. It also showed that an increase in the number of nodes in a system using SD-voting will not result in much reduction in response time because the load is not shared effectively. Systems using the grid protocol have higher maximum throughput and lower response time. Notice that in the MD-voting method, operations are unaware of the topology and the position of the nodes in the grid.

6 Partially Replicated Data

6.1 Background

Fragmentation is a technique where a file is divided into fragments and different fragments may be stored on different nodes. Fragmentation may be necessary because the amount of storage space needed to store a file exceeds the capacity of a node. Also, transactions at a particular node often access a specific portion of the data and storing the frequently accessed part locally will effectively reduce network traffic and delay.

Fragmented data can also be replicated and the fragments can be replicated a different

number of times and stored at a different set of nodes. A single node may not necessarily hold all the fragments of a file. The schemes that can be used to maintain data consistency in such environment are called *partial replication* methods since nodes may maintain only a fraction of the file. Partial replication complicates access to the file because subsets of nodes must be identified that will constitute a complete and current copy.

6.2 Maintaining Partially Replicated Data

A scheme that can be used for maintaining the consistency of partially replicated data is described in [17]. The system consists of N nodes and to reduce storage, only M replicas ($M < N$) are distributed among the nodes. Each replica is subdivided into N fragments, where f_{ij} denotes the j^{th} fragment of the i^{th} replica, $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$. Each node stores M distinct fragments of the file. The fragments stored at a particular node are identified as a *segment*. The *Full Copy Equivalent* (FCE) of a file is the least number of segments necessary in *the worst case* to reconstruct the file. As an illustration, we consider a five-node system with three replicas. Each replica is subdivided into five fragments and the fragments are stored as shown in Figure 4. Note that the segments in nodes 1 and 3 are sufficient to reconstruct the entire replica. However, the segments in nodes 1 and 2 are not sufficient. Note also that any three segments are guaranteed to contain enough fragments to reconstruct the replica. Thus, in the worst case, any three segments are required and FCE = 3. In general, the FCE in the distribution scheme in [17], is equal to $N - M + 1$.

Node 1	Node 2	Node 3	Node 4	Node 5
f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
f_{22}	f_{23}	f_{24}	f_{25}	f_{21}
f_{33}	f_{34}	f_{35}	f_{31}	f_{32}

Figure 4: Distribution 1 of fragments over the nodes

The read and write quorums r and w must satisfy the following constraints:

- $N - M + 1 \leq r \leq N$
- $\max(N - M + 1, \lceil \frac{N+1}{2} \rceil) \leq w \leq N$

- $N + (N - M + 1) \leq r + w \leq 2N$

The read and write quorum combinations that can be used for the system in Figure 4 are $(r = 3, w = 5)$, $(r = 4, w = 4)$ and $(r = 5, w = 3)$. For instance, if $(r = 3, w = 5)$, then the read and write quorum sets R_1 and W_1 are given by:

$$\begin{aligned} R_1 &= \{\{123\},\{124\},\{125\},\{134\},\{135\},\{145\},\{234\},\{235\},\{245\},\{345\}\} \\ W_1 &= \{\{12345\}\} \end{aligned}$$

Note that this particular quorum selection $(r = 3, w = 5)$ is the most desirable from a performance point of view if read operations are predominant.

6.3 Using MD-Voting for Maintaining Partially Replicated Data

Consider the distribution of fragments given in Figure 4. For a read operation, it is possible to access a *smaller* group of segments than the ones in R_1 to reconstruct the complete replica. For instance, $R_2 = \{\{13\},\{14\},\{24\},\{25\},\{35\}\}$ can be used as the read quorum set when the write quorum set $W_2 = W_1 = \{\{12345\}\}$. Since each group of nodes that constitute a quorum group in R_1 is also a quorum group in R_2, R_2 provides better availability than R_1. This is due to the fact that if the file can be read using R_1, it can also be read when groups in R_2 are used. (R_2 is said to *dominate* R_1 [3] since each group in R_1 is a superset of some group in R_2.) The reason the more desirable quorum sets are not possible in [17] is because R_2 is not vote assignable using traditional SD-voting. It is however possible to obtain R_2 and W_2 above using MD(5,5)-voting and the vote and quorum assignments in Table 6.

$$V_{55} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}, \underline{r}_5 = (1,1,1,1,1) \text{ and } \underline{w}_5 = (3,3,3,3,3)$$

Table 6: An MD(5,5)-vote and quorum assignment for partially replicated data

The above observation forms the basis of our partially replicated data scheme described below which is based on MD-voting. Assuming that each file is subdivided into k fragments (the scheme in [17] uses $k = N$):

- Assign the vote u_{ij}, a positive integer, to fragment f_{ij}, for $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, k$, and define the vote assignment V_{Nk} as follows:
  - $v_{aj} = u_{ij}$, if the segment at node a, $a = 1, 2, \ldots, N$, contains the fragment f_{ij} for some $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, k$.
  - $v_{aj} = 0$, otherwise.
- For each dimension j, $j = 1, 2, \ldots, k$, we define the quorums r_j and w_j such that $r_j > 0$, $w_j \geq \left\lceil \frac{\sum_{i=1}^{N} v_{ij}+1}{2} \right\rceil$ and $r_j + w_j = \sum_{i=1}^{N} v_{ij} + 1$.
- Read and write accesses to the partially replicated file use the MD(k, k)-voting protocol and the read and write quorum sets used are:

$$R = \{G \mid G \text{ is a minimal group in MD}(k,k)\text{-voting with respect to } \underline{r_k}\}$$
$$W = \{H \mid H \text{ is a minimal group in MD}(k,k)\text{-voting with respect to } \underline{w_k}\}$$

The following lemmas show that R and W defined above will correctly synchronize read and write operations, and read operations will return the most recent copy of the data.

Lemma 6.2:
Let:

$$R_\ell = \{G \mid G \text{ is a minimal group in MD}(\ell,k)\text{-voting with respect to } \underline{r_k}\}$$
$$W_\ell = \{H \mid H \text{ is a minimal group in MD}(\ell,k)\text{-voting with respect to } \underline{w_k}\}$$

If $\ell \geq \left\lceil \frac{k+1}{2} \right\rceil$, then:

$$\forall G \in R_\ell,\ H \in W_\ell :\ G \cap H \neq \phi$$

and

$$\forall G, H \in W_\ell :\ G \cap H \neq \phi$$

Proof:

Let $G \in R_\ell$ and $H \in W_\ell$, then G and H satisfy the vote requirements in at least ℓ dimensions with respect to $\underline{r_k}$ and $\underline{w_k}$, respectively. Since $\ell \geq \left\lceil \frac{k+1}{2} \right\rceil$, there is at least one dimension j such that $\sum_{a \in G} v_{aj} \geq r_j$ and $\sum_{a \in H} v_{aj} \geq w_j$. Since $r_j + w_j > \sum_{i=1}^{N} v_{ij}$, we have that $G \cap H \neq \phi$.

If $G, H \in W_\ell$, then we have that $\sum_{a \in G} v_{aj} \geq w_j$ and $\sum_{a \in H} v_{aj} \geq w_j$, for some dimension j. Since $2w_j > \sum_{i=1}^{N} v_{ij}$, we have that $G \cap H \neq \phi$.□

From Lemma 6.2 and the fact that $k \geq \left\lceil \frac{k+1}{2} \right\rceil$, we have that:

$$\forall\, G \in R,\ H \in W:\ G \cap H \neq \phi$$

and

$$\forall\, G, H \in W:\ G \cap H \neq \phi$$

The following lemma guarantees that a read operation access a complete replica.

Lemma 6.3:

Each read operation will access a complete replica.

Proof:

Let $G \in R$ and $H \in W$. Since R and W satisfy read and write quorum requirements in all k dimensions, we have that $\sum_{a \in G} v_{aj} \geq r_j$ and $\sum_{a \in H} v_{aj} \geq w_j$. for dimension $j = 1, 2, \ldots, k$. Since $r_j + w_j > \sum_{i=1}^{N} v_{ij}$, each read quorum group contains a fragment j that has been updated by the most recent write operation, for $j = 1, 2, \ldots, k$. Hence, the read operation will obtain the most recent copy of data.□

6.4 Properties of the MD-Voting Approach

The MD-voting scheme assigns votes to fragments in contrast to the scheme presented in [17] which assigns votes to segments. The finer granularity of vote assignment in the MD-voting method allows the system to recognize more available system states in some instances. This has been demonstrated in our previous example (see Section 6.3) where through the use of

MD-voting one can obtain a better read quorum set. This property is formalized in the following lemma.

Lemma 6.4:

For the MD-voting partial replication scheme described in Section 6.3, let R be the read quorum set obtained by setting $u_{ij} = 1$ and $r_j = 1$, for $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, k$. Let G be an arbitrary group of nodes that contains a complete replica. Then,

$$\exists\, H \in R: \quad H \subseteq G$$

Proof:

In order to construct a complete replica, the segments in G must contain a fragment f_{ij}, for each $j = 1, 2, \ldots, k$ and for some $i = 1, 2, \ldots, M$. Then the sum of votes $\sum_{a \in G} v_{aj} \geq 1$, for $j = 1, 2, \ldots, k$ and hence G or a subset of G is a group in R.□

We know from Lemma 6.4 that with MD-voting, it is possible to identify the *minimal* set of segments that constitute a complete copy of data and allow the system to access the data when minimal groups of nodes are operational. In contrast with [17] where a voting scheme is used to identify the set of segments, it is possible to first find the set of of minimal groups of segments and *then* construct an MD-vote assignment to represent it.

Another advantage of using MD-voting for maintaining partially replicated data is *greater flexibility.* Different fragments can be replicated a different number of times and a node can store an arbitrary number of fragments. Distribution of fragments of the replicas is not subject to any constraint which is not true for the scheme in [17]. The correct operation of the protocol is still guaranteed through Lemmas 6.3 and 6.4.

For instance, we can fragment three replicas into four fragments each and distribute the fragments over five nodes in the manner given in Figure 5. The votes assigned to different fragments can be different. For instance, in Table 7, we allocate fragments at node 1 two votes each and fragments at others nodes receive one vote each. Using the quorum assignments given in the figure, the corresponding read and write quorum sets are {{124}, {125}, {145}, {2345}} and {{124}, {125}, {1345}} respectively.

Choosing the optimal fragment distribution, fragment vote and quorum assignment

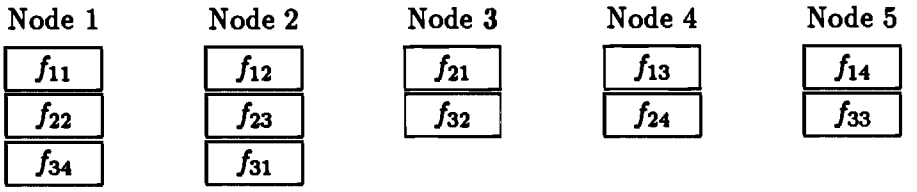


Figure 5: Distribution 2 of fragments over the nodes

$$V_{54} = \begin{pmatrix} 2 & 2 & 0 & 2 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \underline{r_4} = (2,2,2,2) \text{ and } \underline{w_4} = (3,3,2,3)$$

Table 7: An MD(4,4)-vote and quorum assignment for partially replicated data

when storage per node and node availabilities are given, is an interesting problem for future research.

7 Nested Multi-Dimensional Voting

In this section we present another representation method for the multi-dimensional voting scheme. Recall that a quorum group in MD(ℓ, k)-voting satisfies the quorum requirements in *any* ℓ of k dimensions. Another multi-dimensional voting method uses an explicit list of groups of dimension indices called *index set* I for which quorum requirements are to be satisfied. Figure 6 shows an example of the voting procedure using this MD-voting method in a system of six nodes. The index set I used is {{1,2}, {1,3}, {1,4}, {2,4}, {3,4}}. Suppose the votes of nodes 1, 2, 3 and 4 are received. The vector sum of votes is determined and compared to the quorum vector. The quorum requirements in the first and fourth dimensions are satisfied. The voting procedure now proceeds to check if {1,4} is an index group or a superset of an index group in I. Since {1,4} is a group in I, the procedure returns successfully. In this technique, the set I represents an explicit listing of

dimension requirements. MD(ℓ, k)-voting is a special case of this MD-voting method since it uses the index set that consists of all subsets of size ℓ of the set $\{1, 2, \ldots, k\}$. Note that each member of I should be a minimal group and I satisfies the minimality property (see Section 3.1).

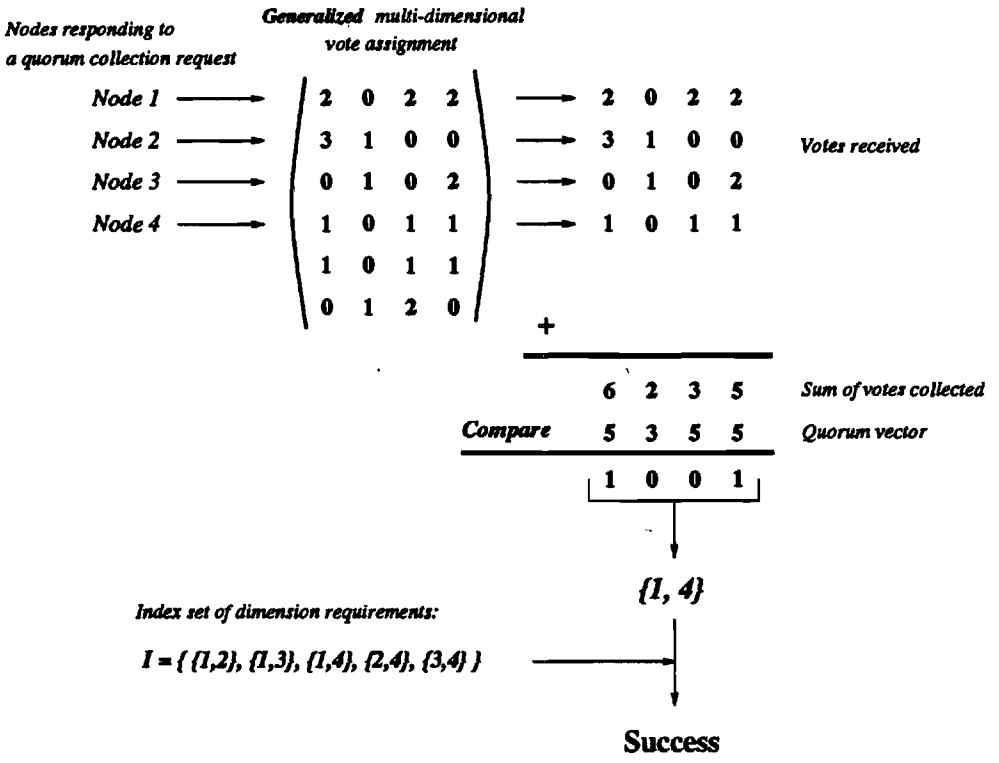


Figure 6: Generalized multi-dimensional voting

A drawback of this new MD-voting scheme is the use of an explicit list of index groups I. Since I is a set of minimal groups under the universe $\{1, 2, \ldots, k\}$ (k is the number of dimensions in the MD-vote assignment), it can be considered as a quorum set for a system of k "nodes". Consequently, we can define I by using MD(ℓ', k')-voting. For example, the index set I in Figure 6 is a quorum set of 4 "nodes" defined by the MD(2,3)-vote and quorum assignment in Table 8.

Figure 7 demonstrates the operation of the system shown previously in Figure 6 with the index set replaced by its MD(2,3) representation. Notice that there are two levels of MD-vote and quorum assignments. At the first level, votes are assigned to the nodes and

$$V_{4,3} = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 1 & 1 \\ 1 & 2 & 1 \end{pmatrix}, \quad \underline{q}_3 = (2,2,2)$$

Table 8: MD-vote and quorum assignment for index set I in Figure 6

the second level vote and quorum assignment is used to represent the index set. First, the votes of nodes 1, 2, 3 and 4 are received and added. The vector sum of votes is then compared to the quorum assignment and it is found that quorum requirement in the first and fourth dimensions are satisfied. Then, the voting procedure uses the index group {1,4} to select the vote values in the second level of the assignment. Rows one and four of the second level vote assignment are added and compared to the quorum requirement of that level. The quorum requirements in the first and second dimensions are satisfied and since the second level uses MD(2,3)-voting, the procedure returns successfully.

Although in the above example the index set I was represented by an MD(2,3)-vote assignment, it could have been represented by MD-voting with another index set I' and so on. For example, I can be represented by the MD-vote assignment

$$V_{4,4} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and index set } I' = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\}$$

Notice that I' itself is representable by an MD(1,1) assignment.[2] We call this multi-level MD-voting technique *nested multi-dimensional* (NMD) voting.

An NMD-voting technique is defined by the number of levels K, a set of multi-dimensional vote and quorum assignments $(V^{(i)}_{N_i,k_i}, \underline{q}^{(i)}_{k_i})$, for $i = 1,2,\ldots K$ and a number ℓ. N_1 is always equal to the number or replicas N and $k_i = N_{i-1}$ for $i = 2,3,\ldots,K$. For example, in Figure 7, the number of levels $K = 2$, the number of nodes at level 1 $N_1 = 6$, the number of "nodes" at level 2 is equal to the number of dimensions at level 1; $N_2 = k_1 = 4$, the number

[2] I' can be obtained by assigning one vote to each node and setting the quorum to two.

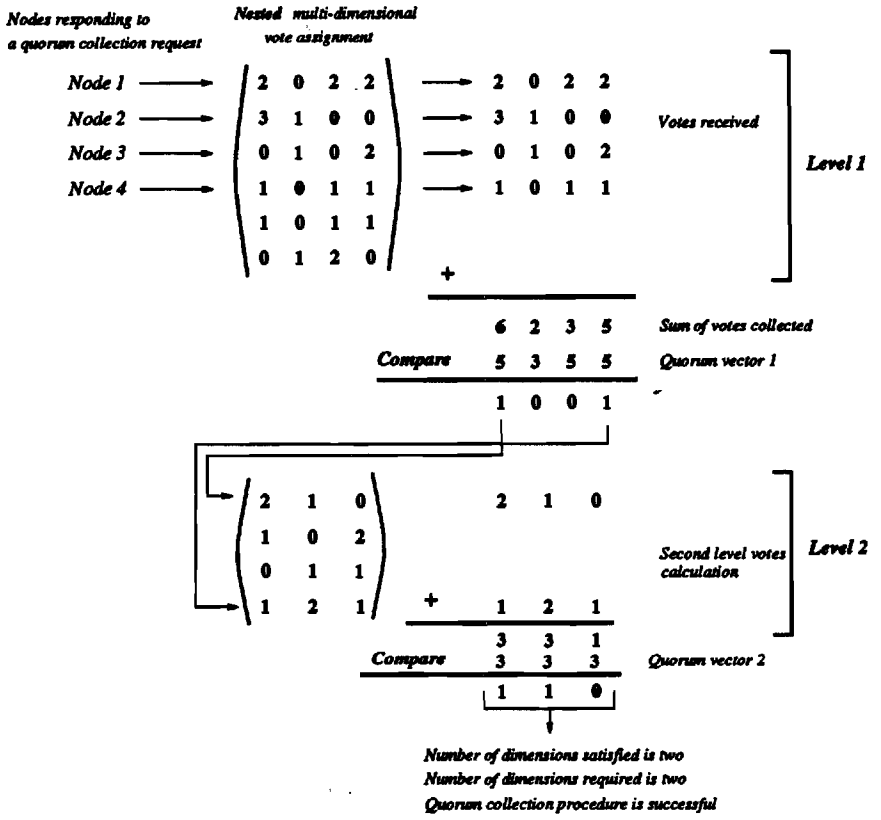


Figure 7: Two level nested multi-dimensional voting with $\ell = 2$ at level 2

of dimensions at level 2 $k_2 = 3$ and $\ell = 2$. The vote and quorum assignments at the two levels $(V_{6,4}^{(1)}, \underline{q}_4^{(1)})$ and $(V_{4,3}^{(2)}, \underline{q}_3^{(2)})$ are shown in Figure 7.

7.1 Example - Hierarchical Quorum Consensus

As an illustration, we present an example of the application of the NMD-voting technique which results in a generalization of the Hierarchical Quorum Consensus (HQC) method presented in [11]. The voting procedure in the HQC method uses a hierarchy of vertices organized into a tree. The highest and lowest levels of the hierarchy contain the root vertex and the leaf vertices, respectively. The leaf vertices are nodes with replicas and constitute the first level of hierarchy (level 0). The non-leaf vertices are logical and they are used only to define the quorum set. The votes from the replicas (at the lowest level) are propagated up the hierarchy along the branches of the tree. A vertex at level i will vote positively if

the number of its child nodes at level $i-1$ that voted positively is greater than or equal to the required quorum. Otherwise it votes negatively. Thus, a read (write) quorum group at the hierarchical level i consists of r_i (w_i) vertices at level $i-1$. The outcome of the voting procedure is determined by the vote of the root vertex.

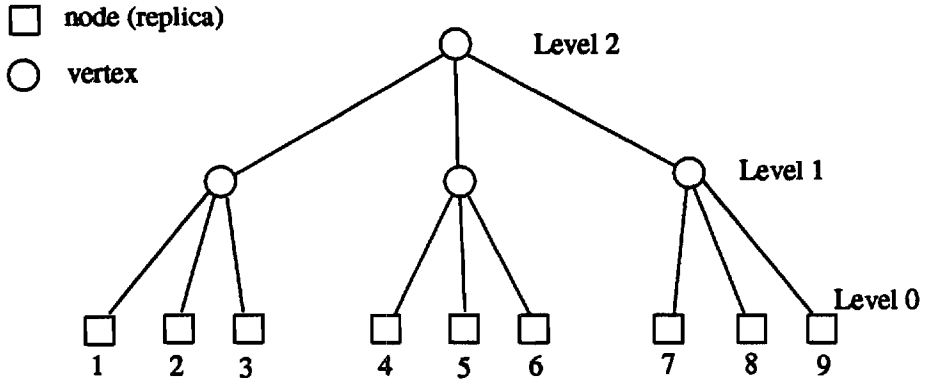


Figure 8: A two-level hierarchical quorum set

As an example, consider the nodes in Figure 8 which are organized into a two-level hierarchy. The read and write quorums at level i are $r_i = 2$ and $w_i = 2$, for $i = 1, 2$. The level 1 quorum sets are:

$$\begin{aligned} R_{21}, W_{21} &= \{\{12\},\{13\},\{23\}\} \\ R_{22}, W_{22} &= \{\{45\},\{46\},\{56\}\} \\ R_{23}, W_{23} &= \{\{78\},\{79\},\{89\}\} \end{aligned}$$

For instance, the left most vertex in level 1 will vote positively if at least two nodes in {1, 2, 3} vote positively, otherwise it votes negatively. The quorum groups at level 2 (the root of the tree) consist of groups from two *different* level 1 quorum sets because the quorum requirement at level 2 is equal to two. The read and write quorum sets defined by the hierarchical structure in Figure 8 are thus equal to:

$$\begin{aligned} Q_2 = \{&\{1245\},\{1246\},\{1256\},\{1345\},\{1346\},\{1356\}, \\ &\{2345\},\{2346\},\{2356\},\{1278\},\{1279\},\{1289\}, \\ &\{1378\},\{1379\},\{1389\},\{2378\},\{2379\},\{2389\}, \\ &\{4578\},\{4579\},\{4589\},\{4678\},\{4679\},\{4689\}, \\ &\{5678\},\{5679\},\{5689\}\} \end{aligned}$$

Each group of Q_2 consists of four nodes which is not a majority group. But notice that any two groups in Q_2 have a non-empty intersection.

$$V_{9,3} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \quad \underline{q}_3 = (2,2,2)$$

Table 9: MD-vote and quorum assignment for a two-level HQC quorum set

Q_2 can be defined using (non-nested) MD(2,3)-voting with the vote and quorum assignments in Table 9. The number of dimensions of MD-vote assignment is equal to the number of hierarchical groups (vertices) at the level 1 which is 3. The vote assignment is derived from the structure of level 0 and 1 with the vote v_{ij} of node i being one if i is a member of hierarchical group j and being zero otherwise. For instance, node 4 is a member of second hierarchical group and correspondingly, its vote assignment is (0, 1, 0). The quorum assignment $\underline{q}$ is the vector (2, 2, 2) and is derived from the fact that for a vertex at level 1 to vote positively, at least two of its children have to return positive replies. The value of ℓ is equal to two because for the vertex at level 2 to return a positive vote, at least two of its children have to vote positively. The MD-vote assignment in Table 9 thus represents the two level hierarchical quorum set defined by the structure in Figure 8 in a natural manner. In general, any quorum set defined by a two-level hierarchy can be represented in a similar manner by the use of MD(ℓ, k)-voting.

Quorum sets that are defined using three or more hierarchical levels do not have a natural MD(ℓ, k)-voting representation and NMD-voting must be used.[3] For instance, Figure 10

[3]We emphasize that one can always find an MD(1,k)-vote and quorum assignment for a HQC quorum set. This will require the list of the quorum groups which may be quite large. For example, the read and

represents a quorum set from a system with 27 replicas which is defined by a three-level hierarchical structure using read and write quorums of two at each level. This hierarchical quorum set can be defined by using a two level NMD-voting with the voting parameters $((V^{(1)}_{27,9}, \underline{q}^{(1)}_9), (V^{(2)}_{9,3}, \underline{q}^{(2)}_3), \ell)$. The vote $v^{(1)}_{ij}$ (the vote assignment to node i in dimension j at level 1) in $V^{(1)}_{27,9}$ is one if node i is a member of hierarchical group j at level 1, for $i = 1, 2, \ldots, 27$ and $j = 1, 2, \ldots 9$, and zero otherwise, and $\underline{q}^{(1)}_9 = (2, 2, ..., 2)$. $V^{(2)}_{9,3}$ and $\underline{q}^{(2)}_3$ are equal to $V_{9,3}$ and $\underline{q}_3$ in Table 9, respectively, and $\ell = 2$.

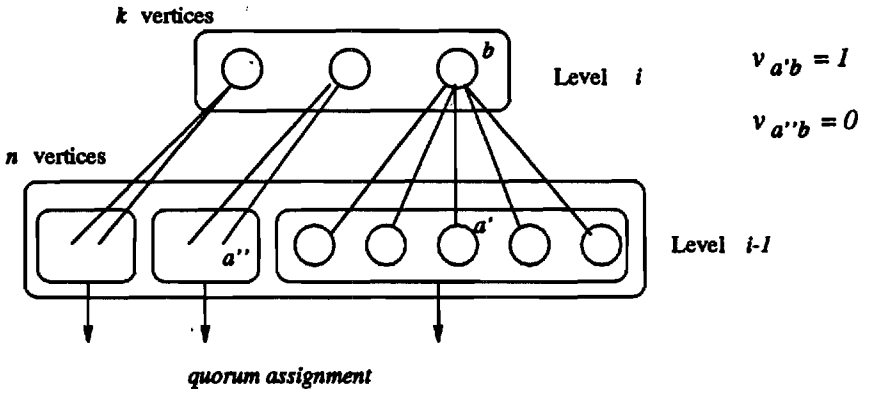


Figure 9: Relationship between the NMD vote and quorum assignment, and the hierarchical structure

The above NMD-vote assignment is obtained by considering the hierarchical structure in Figure 10. Each level of MD-vote and quorum assignment represents one level of hierarchy and the topmost level is represented by the parameter ℓ. For the hierarchical level i, except for the highest level, the number of rows and columns in the MD-vote assignment is equal to the number of vertices at levels $i-1$ and i, respectively. The sizes of the quorum groups at level $i-1$ form the quorum vector for level i (see Figure 9). For instance, to represent the level 1 hierarchical quorum sets defined by the structure in Figure 10 we use a vote assignment with 27 rows (number of nodes at level 0) and nine columns (number of nodes at level 1). The vote value $v^{(i)}_{ab} = 1$ if vertex a at level $i-1$ is a child of vertex b at level i, and otherwise $v_{ab} = 0$. The quorum vector at level i is derived from the quorum required (number of children voting positively) for a node at level i to vote positively. For example, if node b requires at least four of its children to return positive votes, then $q^{(i)}_b$ (the quorum requirement at level i in the b^{th} dimension) will be equal to four. Finally, the parameter ℓ

write quorum sets for the system in Figure 10 contain 81 groups of 8 nodes each.

is set to the quorum required at the root vertex.

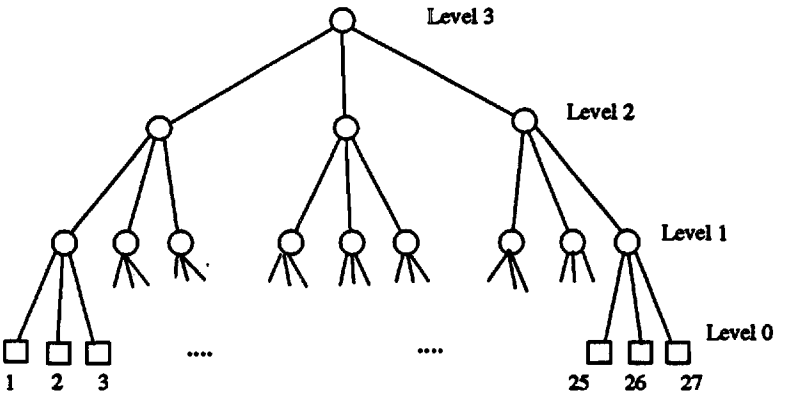


Figure 10: A three-level hierarchical quorum set

One of the advantages of the use of NMD-voting in representing HQC quorum sets is flexibility. We can define more general hierarchical structures where a quorum group at level i consists of subgroups of level $i-1$ of various sizes, i.e., different subgroups of level $i-1$ can use a different quorum. For instance, the quorum set in Figure 11 is defined by a three-level hierarchical structure where all subtrees do not have the same height and all nodes do not have the same number of children. Assuming that each level in the hierarchy uses the majority quorum, then the quorum set defined by the hierarchical structure can be represented by the NMD-vote assignment in Table 10. The NMD-vote and quorum assignment is obtained by applying the technique described above after extending the hierarchical structure so that each node is a leaf at level 3.

With the NMD-voting technique, determining whether a group of nodes constitutes a quorum group is similar to weighted voting. In particular, the quorum collection method, i.e., querying nodes for their votes, can be quite general and is not confined to the polling-like approach described in [11].

8 Concluding Remarks

In this paper, we have introduced the concept of a multi-dimensional vote and quorum assignment which is a generalization of standard voting. In multi-dimensional voting, the vote assigned to a node and the quorum assignment are vectors of non-negative integers

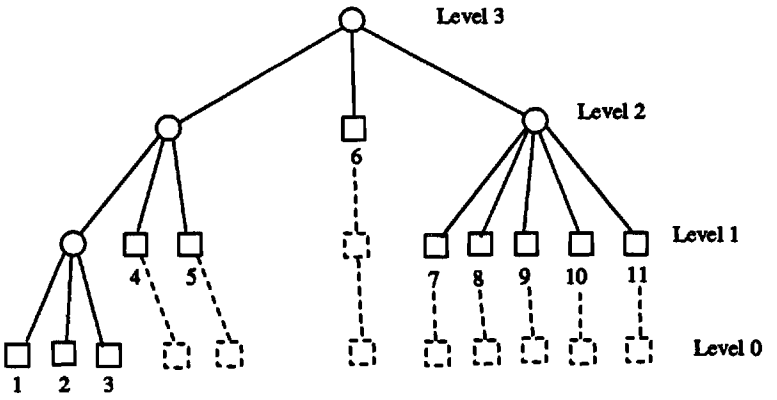


Figure 11: A general three-level hierarchical quorum set

and each dimension is similar to standard voting. We have shown that any set of minimal groups can be represented by multi-dimensional voting and thus MD-voting is as powerful as the quorum set concept, which is the general approach for achieving mutual exclusion in distributed systems. Multi-dimensional voting has the advantage that it is flexible and can be easily implemented.

We have developed an efficient algorithm for finding a multi-dimensional vote and quorum assignment for any set of minimal groups and its use was demonstrated by finding multi-dimensional vote assignments for some non SD-vote assignable quorum sets. We described distributed synchronization methods based on multi-dimensional voting which are easier to implement and/or are more flexible than existing schemes for the same purpose. For example, the multi-dimensional voting scheme for maintaining partially replicated data has no restrictions on the placement of the fragments and a node can store an arbitrary number of fragments. Finally, we presented the nested multi-dimensional voting technique, which is a generalization of multi-dimensional voting, and showed that it may be better suited for representing structured quorum sets in some instances.

We have demonstrated that MD-voting can indeed be used to implement a wide range of replica control protocols. Future research needs to consider the performance implications of the additional flexibility provided through the use of MD-voting.

$$V_{11,9}^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \underline{q}_9^{(1)} = (2,1,1,1,1,1,1,1,1)$$

$$V_{9,3}^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \quad \underline{q}_3^{(2)} = (2,1,3), \quad \text{and } \ell = 2$$

Table 10: MD-vote and quorum assignment for the three-level HQC quorum set in Figure 11

References

[1] S. Davidson, H. Garcia–Molina, and D. Skeen, "Consistency in partitioned network," *ACM Computing Survey*, vol. 17, no. 3, pp. 341–370, 1985.

[2] M. Ahamad and M. Ammar, "Performance characterization of quorum-consensus algorithms for replicated data," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 492–496, 1989.

[3] D. Barbara and H. Garcia–Molina, "Mutual exclusion in partitioned distributed systems," *Distributed Computing*, vol. 1, pp. 119–132, 1986.

[4] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, pp. 95–114, 1978.

[5] H. Garcia–Molina and D. Barbara, "How to assign votes in a distributed system," *Journal of ACM*, vol. 32, no. 4, pp. 841–860, 1985.

[6] H. Gifford, "Weighted voting for replicated data," in *Proceedings of 7th Symposium on Operating Systems*, pp. 150–162, ACM, 1979.

[7] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems*, vol. 4, pp. 180–209, June 1979.

[8] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 145–159, May 1985.

[9] D. Agrawal and A. El Abbadi, "An efficient solution to the distributed mutual exclusion problem," in *Proceedings of Principles of Distributed Computing*, pp. 193–200, ACM, 1989.

[10] S. Y. Cheung, M. H. Ammar, and M. Ahamad, "The grid protocol: A high performance scheme for maintaining replicated data," in *Proceedings of 6th International Conference on Data Engineering*, pp. 438–445, IEEE, 1990.

[11] A. Kumar, "Performance analysis of a hierarchical quorum consensus algorithm for replicated objects," in *Proceedings of 10th International Conference on Distributed Computing Systems*, pp. 378–385, IEEE, 1990.

[12] D. Eager and K. Sevcik, "Achieving robustness in distributed database systems," *ACM Transactions on Database Systems*, vol. 8, no. 3, pp. 354–381, 1983.

[13] M. Herlihy, "Dynamic quorum adjustment for partitioned data," *ACM Transactions on Database Systems*, vol. 12, pp. 170–194, June 1987.

[14] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 230–280, 1990.

[15] D. Barbara, H. Garcia–Molina, and A. Spauster, "Increasing availability under mutual exclusion constraints with dynamic vote reassignment," *ACM Transcations on Computer Systems*, vol. 7, pp. 394–426, Nov 1989.

[16] M. Ahamad, M. H. Ammar, and S. Y. Cheung, "Optimizing the performance of quorum consensus replica control protocols," in *Proceedings of the Workshop on Management of Replicated Data*, pp. 102–107, IEEE, 1990.

[17] D. Agrawal and A. El Abbadi, "Reducing storage for quorum consensus algorithms," in *Proceedings of Very Large Databases Conference*, pp. 419–430, 1988.

[18] S. Y. Cheung, M. Ahamad, and M. H. Ammar, "Optimizing vote and quorum assignments for reading and writing replicated data," *IEEE Transactions on Knowlegde and Data Engineering*, vol. 1, pp. 387–397, September 1989.

[19] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

Appendix A: Algorithm for Finding an MD-vote assignment

In [18], a technique is described for testing if a set of groups Q is SD-vote assignable. The linear program LP(Q) shown in Figure (12) is set up using the groups in Q. The

$$\begin{aligned}
\text{Minimize:} \quad & \sum_{i=1}^{N} v_i + q \\
\text{s.t.} \quad & \forall\, G \in Q: \ \sum_{g \in G} v_g \geq q && (2) \\
& \forall\, H \in \overline{sup}(Q \,|\, U) \cup psub(Q): \ \sum_{h \in H} v_h \leq q - 1 && (3) \\
& v_i \geq 0, \ \ i = 1, 2, \ldots, N \\
& q \geq 1
\end{aligned}$$

Figure 12: LP(Q): Linear program corresponding to Q

following notations are used:

- $\overline{sup}(Q \,|\, U)$ is the set of all groups that are subsets of U ($= \{1, 2, \ldots, N\}$) and not supersets of any group in Q, and
- $psub(Q)$ is the set of all groups that are proper subsets of the groups in Q.

If LP(Q) does not have a feasible solution then Q is not SD-vote assignable. Otherwise a rational solution is found and can be converted to an integral vote and quorum assignment.

Unlike [18], in this paper we are dealing with quorum sets Q that have the minimality property. This allows us a further refinement of LP(Q) which is a result of the following lemma.

Lemma A.1: Let Q be a set of groups satisfying the minimality property, i.e., $\forall\, G, H \in Q: \ G \not\subseteq H$. Then,

$$psub(Q) \subseteq \overline{sup}(Q \,|\, U)$$

Proof: The proof is by contradiction.

Assume there is a group X that is in $psub(Q)$ but not in $\overline{sup}(Q \mid U)$. Since $X \in psub(Q)$, X is a proper subset of some group $A \in Q$. Furthermore, $X \notin \overline{sup}(Q \mid U)$ so it is a superset of some (other) group B such that $B \in Q$. However, then Q would violate the minimality property because the above facts imply that $B \subset A$. This contradicts the premise.□

We can thus substitute constrain (3) in LP(Q) with,

$$\forall\, H \in \overline{sup}(Q \mid U): \quad \sum_{h \in H} v_h \leq q - 1 \tag{4}$$

We extend the SD-vote finding procedure to find an MD(1,k) vote and quorum assignment for a quorum set Q. The algorithm (illustrated in Figure 2) constructs an MD(1,k)-vote assignment by testing to see if Q is SD-vote assignable. If not, groups are systematically removed from Q until the groups that remain form an SD-vote assignable quorum set. The votes and quorum obtained from the solution form the assignment in the first dimension. The set of groups removed from Q to make it SD-vote assignable are then used as input to a second iteration to find the second dimension of vote and quorum assignment. This is repeated until all groups are represented by the MD-vote assignment. Since a quorum set with a single group is SD-vote assignable, in each iteration at least one group of Q is removed, and the algorithm is guaranteed to terminate.

In what follows, we address the following two elements of the MD-vote finding algorithm in more detail.

1. The use of the Simplex tableau of LP(Q) to decide which group to remove when LP(Q) is infeasible.
2. A method by which the effort expended to determine that LP(Q) is not feasible is used to facilitate the solution of LP($Q - \{A\}$) when group A is removed from Q.

A.1 Choosing a Group for Removal

In the first phase of the Simplex method, artificial variables are added to the constraints (2) of LP(Q) to obtain a basis and each of these constraints correspond to one quorum group in Q. These artificial variables are pivoted out of the basis during this phase (i.e.,

the artificial variables are eliminated) and if this cannot be achieved, it is an indication that Q is not SD-vote assignable. In this case, a row that contains a basic artificial variable is selected and removed. Since only the rows that correspond to groups of Q contain artificial variables, we will always remove a constraint that corresponds to a group $A \in Q$. If there are several rows with basic artificial variables, one is chosen arbitrarily for removal.

A.2 Reusing Computational Effort Spent in Solving LP(Q)

In the MD-vote finding algorithm, several related linear programs may need to be constructed and solved until a feasible solution is found. Since the linear programs are derived from one another, the computational effort expended to detect feasibility of one linear program can be used in the next and need not be wasted. The removal of a group A from Q will result in the deletion of one constraint from (2), namely,

$$\sum_{g \in A} v_g \geq q \tag{5}$$

However, since the set of constraints (4) is derived from $\overline{sup}(Q \mid U)$, the removal of a group from Q will cause significant changes in this set. The new linear program to determine if the quorum set $Q - \{A\}$ is SD-vote assignable is LP($Q - \{A\}$) given in Figure 13. The set $\overline{sup}(Q \mid U)$ is a proper subset of $\overline{sup}(Q - \{A\} \mid U)$, this is because any group that is not a superset of groups of Q is also not a superset of groups of $Q - \{A\}$. Also, A is a group of $\overline{sup}(Q - \{A\} \mid U)$ but not a group of $\overline{sup}(Q \mid U)$. The set $[\overline{sup}(Q - \{A\} \mid U) - \overline{sup}(Q \mid U)]$, which represents the new constraints in LP($Q - \{A\}$), contains only groups that are supersets of group A. Therefore, LP($Q - \{A\}$) has one less constraint (namely constraint (5)) and several new constraints that are derived from groups that are supersets of A.

We derive a lemma that shows how to obtain the set of constraints that must be added to LP(Q) to obtain LP($Q - \{A\}$) so that the solution procedure (Simplex method) can be continued from the point where it was detected that LP(Q) was not feasible. Let $Q \backslash A$ be the set of groups obtained from Q by taking the difference of each group of Q and the group A, i.e.,

$$Q \backslash A = \{(G - A) \mid G \in Q\}$$

For instance, let $Q = \{\{1,2\},\{1,3\},\{2,3\}\}$ and $A = \{3\}$, then $Q \backslash A = \{\{1,2\},\{1\},\{2\}\}$. The following lemma gives the groups that must be added to LP(Q) to obtain LP($Q - \{A\}$):

$$
\begin{aligned}
\text{Minimize:} \quad & \sum_{i=1}^{N} v_i + q \\
\text{s.t.} \quad & \forall\, G \in Q - \{A\} : \sum_{g \in G} v_g \geq q && (6) \\
& \forall\, H \in \overline{sup}(Q - \{A\} \,|\, U) \ \sum_{h \in H} v_h \leq q - 1 && (7) \\
& v_i \geq 0, \ \ i = 1, 2, \ldots, N \\
& q \geq 1
\end{aligned}
$$

Figure 13: LP($Q - \{A\}$): Linear program corresponding to $Q - \{A\}$

Lemma A.1: Let Q be a set of groups that satisfy the minimality property, then,

$$\overline{sup}(Q - \{A\} \,|\, U) = \overline{sup}(Q \,|\, U) \cup \{G \cup A \mid G \in \overline{sup}(((Q - \{A\})\backslash A) \,|\, U - A)\}$$

Before we can show Lemma A.1, we need to show the following auxiliary lemma.

Lemma A.2: Let Q be a set of groups that satisfy the minimality property. If $G \in \overline{sup}((Q - \{A\})\backslash A \,|\, U - A)$, then $G \cup A \in \overline{sup}(Q - \{A\} \,|\, U)$.

Proof: The proof is by contradiction.

Each group $G \in \overline{sup}((Q - \{A\})\backslash A \,|\, U - A)$ is a subset of $U - A$ and not a superset of any group of $(Q - \{A\})\backslash A$. Suppose there exists a group G such that it is a group in $\overline{sup}((Q - \{A\})\backslash A \,|\, U - A)$ and $G \cup A$ is not a group in $\overline{sup}(Q - \{A\} \,|\, U)$. Then it must be that $G \cup A$ is a superset of some group $B \in Q - \{A\}$ and thus,

$$
\begin{aligned}
B \subset G \cup A \ &\Longrightarrow \ B - A \subset G \\
&\Longrightarrow \ G \text{ is a superset of } B - A
\end{aligned}
$$

Since $B - A$ is a group of $(Q - \{A\})\backslash A$, this implies that $G \notin \overline{sup}((Q - \{A\})\backslash A \,|\, U - A)$ and it contradicts the fact that $G \in \overline{sup}((Q - \{A\})\backslash A \,|\, U - A)$.□

The set $\overline{sup}((Q - \{A\})\backslash A \mid U - A)$ is the set of groups that are subsets of $U - A$ and not supersets of any group of $(Q - \{A\})\backslash A$. Lemma A.2 states that when we add A to a group $G \in \overline{sup}((Q - \{A\})\backslash A \mid U - A)$, the resulting group $G \cup A$ will not be a superset of any group of $Q - \{A\}$ under the universe of nodes U. These are precisely the groups that will be added to LP(Q) to obtain LP($Q - \{A\}$) which was given in Lemma A.1 above. Now we can show the correctness of Lemma A.1.

Proof of Lemma A.1: We will show that LHS $\subseteq$ RHS and RHS $\subseteq$ LHS.

Let X be an arbitrary group in $\overline{sup}(Q - \{A\} \mid U)$. If $X \in \overline{sup}(Q \mid U)$, then LHS $\subseteq$ RHS is trivially true. Let $X \notin \overline{sup}(Q \mid U)$, then it must be that X is a superset of A and X is not a superset of any group in $Q - \{A\}$. We can write X as,

$$X = Y \cup A$$

where Y is not a superset of any group in $(Q - \{A\})\backslash A$ under the universe $U - A$. Thus, X which is equal to $Y \cup A$ is a member of $\{G \cup A \mid G \in \overline{sup}((Q - \{A\})\backslash A \mid U - A)\}$ because $Y \in \overline{sup}((Q - \{A\})\backslash A \mid U - A)$ and hence LHS $\subseteq$ RHS.

Since $\overline{sup}(Q \mid U) \subseteq \overline{sup}(Q - \{A\} \mid U)$, we can show that RHS $\subseteq$ LHS by showing $\{G \cup A \mid G \in \overline{sup}((Q - \{A\})\backslash A \mid U - A) \subseteq \overline{sup}(Q - \{A\} \mid U)$. Let X be an arbitrary group in $\{G \cup A \mid G \in \overline{sup}((Q - \{A\})\backslash A \mid U - A)$, then X can be written as,

$$X = Y \cup A$$

for some $Y \in \overline{sup}((Q - \{A\})\backslash A \mid U - A)$. By Lemma A.2, we have $X \in \overline{sup}(Q - \{A\} \mid U)$.□

The constraints that must be added when A is removed are derived from groups of the set $\{G \cup A \mid G \in \overline{sup}((Q - \{A\})\backslash A \mid U - A)\}$. Notice that this set always contains A as an element because the empty set is an element of $\overline{sup}((Q - \{A\})\backslash A \mid U - A)$. Lemma A.1 thus defines the set of constraints that are added to LP(Q) to obtain LP($Q - \{A\}$) when group A is removed from Q and provides us with a method to convert LP(Q) to LP($Q - \{A\}$).

Distributed Computing (1990) 4:69–80

DISTRIBUTED COMPUTING

Fault-tolerant atomic computations in an object-based distributed system*

Mustaque Ahamad, Partha Dasgupta, and Richard J. LeBlanc, Jr.

School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA

Received December 22, 1988/Accepted February 20, 1990

Mustaque Ahamad received his B.E. (Hons.) degree in Electrical Engineering from the Birla Institute of Technology and Science, Pilani, India. He obtained his M.S. and Ph.D. degrees in Computer Science from the State University of New York at Stony Brook in 1983 and 1985 respectively. Since September 1985, he is an Assistant Professor in the School of Information and Computer Science at the Georgia Institute of Technology, Atlanta. His research interests include distributed operating systems, distributed algorithms, fault-tolerant systems and performance evaluation.

Partha Dasgupta is an Assistant Professor at Georgia Tech since 1984. He has a Ph.D. in Computer Science from the State University of New York at Stony Brook. He is the technical project director of the Clouds distributed operating systems project, as well as a co-principal investigator of Georgia Tech's NSF-CER award. His research interests include building distributed operating systems, distributed algorithms, fault-tolerant systems and distributed programming support.

Richard J. LeBlanc, Jr. received the B.S. degree in physics from Louisiana State University in 1972 and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin – Madison in 1974 and 1977, respectively. He is currently a Professor in the School of Information and Computer Science of the Georgia Institute of Technology. His research interests include programming language design and implementation, programming environments, and software engineering. Dr. LeBlanc's current research work involves application of these interests in distributed processing systems. As co-director of the Clouds Project, he is studying language concepts and software engineering methodology for utilizing a highly reliable, object-based distributed system. He is also interested in specification-based software development methodologies and tools. Dr. LeBlanc is a member of the Association for Computing Machinery, the IEEE Computer Society and Sigma Xi.

Abstract. A distributed system can support fault-tolerant applications by replicating data and computation at nodes that have independent failure modes. We present a scheme called parallel execution threads (PET) which can be used to implement fault-tolerant computations in an object-based distributed system. In a system that replicates objects, the PET scheme can be used to replicate a computation by creating a number of parallel threads which execute with different replicas of the invoked objects. A computation can be completed successfully if at least one thread does not encounter any failed nodes and its completion preserves the consistency of the objects. The PET scheme can tolerate failures that occur during the execution of the computation as long as all threads are not affected by the failures. We present the algorithms required to implement the PET scheme and also address some performance issues.

Key words: Fault-tolerant computing – Atomicity – Distributed systems and replication

* This work was supported in part by NSF grants CCR-8619886 and CCR-8806358, and RADC contract number F30602-86-C-0032

Offprint requests to: M. Ahamad

1 Introduction

Distributed computing systems offer many advantages which include resource sharing, parallelism and the potential for increased availability due to multiplicity of components. A distributed implementation of an application can exploit the concurrency available in the application by executing parts of the computation at different nodes. However, it is possible that some components of the distributed system (nodes or communication links) fail while others remain operational. This could result in an inconsistent system state where the results of a computation are reflected at some nodes but not at others. *Atomic actions* (we refer to them simply as actions) provide a mechanism which guarantees that a computation either completes at all nodes or it has no effect on the state of the system.

The atomicity property provided by actions masks failures from users by undoing a partially completed computation when a failure is detected. This ensures that the system will remain in a consistent state; however, actions do not promise *forward progress*. Thus, though failures do not cause inconsistent executions, they can lead to repeated aborts of an action. Although the failure of an arbitrary set of system components can make progress impossible, we would like to guarantee that a computation, once started, will complete when not more than a certain number of node or communication failures occur in the system. The failures may occur either before the computation was started or during its execution.

If a data item is replicated at a number of nodes with independent failure modes then an action that needs to access the item can make progress even when some of the nodes where the item is stored have failed. However, care must be taken to ensure the consistency of such data replicas. Many consistency maintenance algorithms have been proposed for use with replicated data [9, 13, 4].

In an object-based system, data is encapsulated in objects which define the operations that can be used to access the data. An action may invoke operations defined by many objects during the course of its execution. When objects are replicated, it must be ensured that any replica touched (read or written) by the action is not outdated. When a node that stores a touched replica fails, the action must be aborted even when nodes having other replicas are operational. In this case, the action can be repeatedly tried until it completes. The failed nodes can be avoided by the system in successive tries. However, this increases the time between the start and completion of the action, which increases the probability of further node failures occurring during its execution. An alternative to the sequential repetition is to replicate the computation of the action. Conceptually, the action computation is executed by many parallel threads (a thread represents an independent execution of the action) and each thread executes as if there is no replication. The action can be completed when at least one thread does not encounter failed nodes and a sufficient number of replicas of each touched object are available. The latter condition is necessary for maintaining consistency. In this paper, we present such a scheme which we call *parallel execution threads* (PET). The set of parallel threads executing on behalf of an action are called a PET computation.

The replication of computation in an object-based system presents problems that do not arise when replicated data is accessed by non-replicated computations. Since objects are accessed by arbitrary operations, in the general case, the execution of an operation on an object can change the state of the object in a non-deterministic fashion. Thus, executing an operation separately at all replicas even with the same input parameters may lead to a situation where the states of the replicas are no longer the same. Such a state can result in inconsistent executions. In addition, the invocation of an operation can lead to more nested invocations. These situations do not arise in maintaining consistency of replicated data (as opposed to objects) because the only operation that updates the data is a write (as is the case when files are replicated). Similar problems do arise when processes that encapsulate data are replicated.

We describe how replication of data and computation is handled in the PET scheme, giving rise to an implementation of fault-tolerant actions in an object-based system. We present the assumptions about the system in Sect. 2. The problem of computation replication is described in Sect. 3. The PET scheme in presented Sect. 4 and the implementation issues related to PET are presented in Sect. 5. Sections 6 and 7 address the correctness and performance of the PET scheme. Related work is described and compared with the PET scheme in Sect. 8. The paper is concluded in Sect. 9.

2 System model

We consider replicated computations in an object-based system. The model used is similar to the one which is provided by the *Clouds* operating system [7]. *Clouds* supports objects and threads. An object is a persistent instance of an abstract data type. The data encapsulated in an object can be accessed by a number of operations defined by the object. Thus, objects provide storage for both data and the code to be executed when computations invoke the objects. To support fault-tolerance, objects may be replicated.

A computation is executed in *Clouds* by one or more threads. A thread is an active entity that can invoke operations defined by a set of objects, which may reside at many nodes. The state of an object is defined by the data encapsulated in it; threads transform the object state by possibly changing the values stored in its data items. We allow non-deterministic operations and hence when the same operation is executed at two replicas of an object with identical states, the resulting states may be different. This has implications on what kind of schemes can be used for implementing replicated computations.

We assume objects to be relatively heavy-weight compared to the notion of objects in the programming language parlance (e.g. Smalltalk). A *Clouds* object is

a long-lived virtual space and the data defined by the object resides permanently in its virtual space until the object is deleted. Objects correspond to entities such as queues, directories, page maps, text files, database relations, and libraries; not to fine-grained entities such as integers. However, an object logically belongs to a single node even though parts of the object may be temporarily moved to other nodes. A large amount of data and/or code that needs to be stored at several nodes should be viewed as a collection of objects.

We will consider two mechanisms for implementing invocations of remote objects:

1. An object is invoked at the node where it exists by a remote procedure call (RPC) mechanism. In this case, conceptually, the caller thread moves to the node where the object is stored and hence the computation is distributed.
2. The invoked object is brought to the invoking node (via demand paging) and computation proceeds at the invoking node. This mechanism can be supported by the distributed shared memory (DSM) abstraction [12] and is useful in a data servers/workstations environment in which the objects are stored at the data servers but the computation must take place at the workstations.

The PET scheme provides fault-tolerance by executing a computation using several replicated threads. It guarantees consistency by using quorum protocols and it uses commit protocols to ensure that changes made by a single thread are reflected at all touched objects. To achieve this, substantial amount of communication overhead is incurred. Although it may be possible to use it in a wide-area network, it is better suited in a local-area network environment where hardware support for multicast can be exploited to reduce the communication overhead. Furthermore, the latency in the DSM approach can become unacceptable in a wide-area network. To keep the description of the algorithms simple, we assume reliable message delivery but it can be easily seen that in many cases, communication failures can be handled in a manner similar to failures of nodes. Node failures are assumed to be *fail-stop*. Furthermore, we assume that when an invoked object is at an operational node, the invocation will return in a fixed time with a high probability. Thus, timeouts are used to deal with node or communication failures. Our scheme can deal with network partitions and we do not require that node and communication failures be distinguishable.

Each object has a permanent copy on stable storage which is updated only when a thread commits. Since the copy of the object in volatile memory is lost when a node fails, a thread that has not copied its state to stable storage is aborted when a node it touches fails.

3 Replicating objects and invocations

In the object/thread model, a thread executes by invoking operations of a set of objects which may be stored at many nodes. When objects are replicated for fault-tolerance, several approaches are possible for executing the computation represented by a thread. For example, a thread can execute with a single replica of an invoked object. In this case, computation is not replicated and it will fail if the thread invokes a replica at a failed node or a node participating in the execution fails before the thread terminates. Such an approach was used by ISIS [5] but to provide fault-tolerance, checkpointing and restart of threads was used. In another approach, computation itself can be replicated whenever a replicated object is invoked. In the CIRCUS approach [6], when a thread invokes an object having n replicas, n threads are created and each thread executes with a separate replica. Thus, the invocation of a replicated object results in replication of computation. When several objects are invoked in a nested manner, this can lead to a large number of threads and a replica can receive multiple invocation requests when a single invocation is to be executed. This requires that nodes collate invocation requests and results returned by them. The ISIS and CIRCUS approaches can be contrasted in the sense that one does not replicate computation while the other replicates computation to the maximum possible degree.

The CIRCUS approach offers the advantage that an invocation of an object can return successfully even when some of the called replicas are at failed nodes. Thus, fault-tolerance can be provided without using other mechanisms such as checkpointing. However, when operations defined by objects are non-deterministic, execution of an operation at each node separately may not work because the same operation invoked at two identical replicas may produce different results. The use of a single thread as in ISIS does not have this problem because execution is done at a single node and the updated state of the replica is copied to other nodes.

Our goal is to provide a method for implementing computations in an object-based system which has the following properties.

1. A thread is a basic entity in the system and its implementation should remain simple and efficient even when objects are replicated.
2. A computation should be able to complete even when failures occur either prior to or during its execution. This can be achieved by using replication, but the system state and the results returned when a computation completes must be the same as in a failure-free system that uses no replication.

We present a scheme that replicates computation to provide fault-tolerance but works even when object operations are non-deterministic. Our approach is based on a synthesis of the methods proposed in [5] and [6]. For each top-level fault-tolerant computation, we create several threads and each thread executes independently as in a non-replicated environment. Thus, a thread invokes only one replica of each replicated object it touches. When one or more threads return successfully, one of them is committed and the rest are aborted. The details of the scheme, the necessary algorithms and performance issues are considered in the following sections.

4 The PET scheme

In the PET scheme, which is intended to be used in a system that replicates objects, a computation is executed by a number of parallel threads. Each thread begins execution by invoking the same operation of an object but further invocations may be different because object operations can be non-deterministic. The execution of a thread proceeds as if there is no replication of objects because a single replica of each invoked object is used by the thread.

We denote by $A_{i,j}(1 \leqq j \leqq n)$ the j^{th} thread of the PET computation A_i. The node where the computation A_i starts is called its *coordinator* and we denote it by $C(A_i)$. $C(A_i)$ decides the number of threads to be created and chooses a node for each thread where its execution begins. We call this node the home of the thread and use $H(A_{i,j})$ to refer to the home node of thread $A_{i,j}$. When $A_{i,j}$ invokes an object, a single replica of the invoked object is chosen (see Sect. 5.1). $A_{i,j}$ executes with a copy of the replica which is obtained from the permanent state of the replica. Also, the thread does not request any locks at this time and hence can execute without delay when the node is operational (locks are obtained when a thread is committed). If all nodes where replicas invoked by $A_{i,j}$ exist are operational and this thread can get access to the replicas, then $A_{i,j}$ completes and returns success to its home node $H(A_{i,j})$. We use $O_{l,r}$ to refer to the r^{th} replica of object O_l.

In Fig. 1, two threads are created for executing the computation A_i which begins execution by invoking an operation defined by object O_1. The replica of O_1 invoked by each thread is chosen from the replicas that exist at nodes which are thought to be operational using a scheme considered in a later section. Since each thread executes as if there is no object replication in the system, no further parallel threads are created when the threads invoke objects O_2 and O_3. In the example, both $A_{i,1}$ and $A_{i,2}$ invoke the same set of objects but they execute

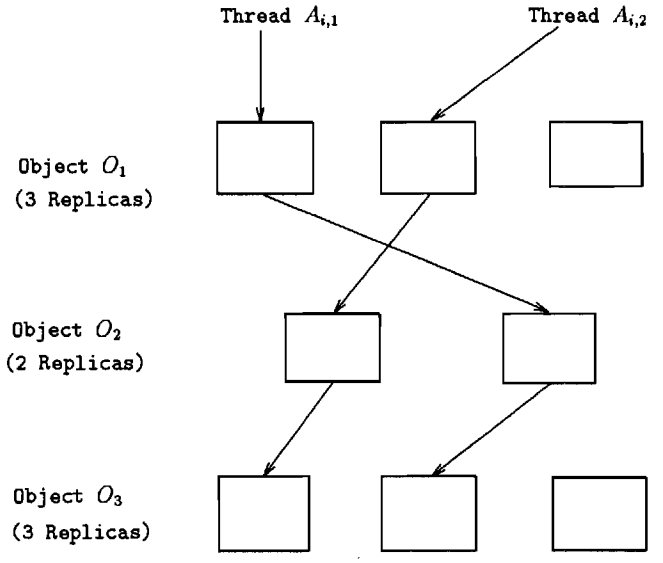


Fig. 1. Parallel execution threads

with different replicas of each object. The latter may not be possible when the scheme used to choose replicas does not know about the replicas that have been invoked by other threads of the same action or when the degree of replication of an invoked object is smaller than the number of threads. In such a case, we allow multiple threads belonging to the same action to invoke a single replica.

Although the PET computation is executed by many threads, its result should be the same as if a single thread executed the computation in a failure-free environment. Since the execution of each thread may not be identical because of non-deterministic operations, the PET computation should be completed by committing a single thread and undoing the effects of all other threads. A thread that returns successfully to its home node may not be able to commit when it is chosen to do so because failures can occur after the thread returns to its home node. To provide fault-tolerance, the commit protocol used by the PET scheme must try to commit threads until it is successful or all threads encounter failed nodes while invoking the replicas or in the commit phase. In the latter case, the PET computation is aborted.

We take an optimistic approach in which at invocation time, a thread only needs to get access to a single replica and not be aware of the other replicas of the object. To preserve consistency of objects, concurrent PET computations must be synchronized with respect to each other. Furthermore, the changes to the state of each object cannot just be applied to a single replica, rather, they have to be reflected at other nodes where replicas of the object exist. We use a quorum-based method [9] for synchronization and copy the updated replica to nodes which store other replicas of the object. Since updates made by a single thread are copied to other nodes, non-deterministic execution does not present any problems. Furthermore, the quorum method ensures consistency even when the network is partitioned.

5 Implementation of PETs

The implementation of the PET scheme requires algorithms for choosing the replicas to be invoked by a thread, for synchronization of the threads, and for propagation of replica states when a thread is committed. Since the execution of each of the threads proceeds as if there is no replication, either RPC or the DSM mechanisms can be used to implement it. When RPC is used, the thread computation is done at each of the nodes where a replica touched by the thread is stored. In the other case, the computation is done at a single node but it uses replicas which are stored in stable storage at other nodes. The algorithms are similar for both mechanisms, any differences will be considered in the following discussion.

5.1 Choosing a replica

When a thread invokes an object, the choice of the replica of the called object for executing the invocation can

affect the number of failures that can be tolerated by the action. If the object has been previously invoked by the thread, the replica that was used for the earlier invocation *must* be chosen for correct execution of the thread. Otherwise, though it is possible to choose any of the replicas that exist at nodes thought to be operational, a better scheme will choose one that permits a greater number of failures to be tolerated. Since the failure of a node where an already touched replica exists will abort the action, it is better to choose a replica at one of these nodes, thus minimizing the number of nodes touched by the thread. If such a replica does not exist then the chosen one should be at a node that is not touched by other threads, thus minimizing the number of threads aborted by the failure of a single node. We propose the following scheme for choosing replicas.

1. If the object was invoked by the thread previously, use the same replica that was used before to execute the invocation.
2. Otherwise, if a replica exists at the node where the thread is currently executing, the invocation is executed using the local replica of the called object.
3. Otherwise, determine if a replica exists at any of the nodes already touched by the thread. If such a node exists then use the replica at that node.
4. Otherwise, choose the replica at a node that is thought to be operational and has been touched by the smallest number of other threads of the same action. (This may not be known accurately and hence the decision must be based on what is known to the caller node. The correctness of the PET scheme does not depend on the accuracy of this knowledge.)

The proposed scheme is applicable in both cases when the computation is distributed (RPC) and also when it is done at a single node (DSM). In the first case, since the computation is distributed at nodes where touched replicas are stored, minimizing the number of such nodes increases the probability that the thread is not aborted due to a failure. In the second case, all replicas are brought at a single node and the computation is done locally. However, the replicas should be stored at their source nodes at the time the thread commits. This is necessary in a workstation/data server environment where the workstations do not have enough stable storage or are diskless. Thus, again, minimizing the number of nodes that participate in the execution of a thread provides a higher degree of fault-tolerance.

5.2 Invocation of a replica by multiple threads

It can be seen that when the above scheme is used to choose a replica, more than one thread of an action may invoke the same replica. Since object operations can be non-deterministic, the calls to the replica can have different parameters and hence cannot be collated; they must be executed separately. For this purpose, we use the following scheme. The two threads are treated as different computations and when a thread invokes the replica for the first time, it is given a copy of the replica state from stable storage which it uses for all calls. We can do this because threads belonging to the same action do not need to communicate or share. Thus, each thread works with its own copy of the replica. When one of these threads is committed, the copies belonging to the aborted threads are discarded.

A number of volatile copies of a replica will be created at a node where a replica is invoked by multiple threads of a PET computation. These copies must be retained until the PET computation terminates because any future invocation by a thread must execute with its volatile copy. The use of techniques such as copy-on-write can reduce the storage cost since only those parts of the replica that are updated need to be stored separately.

5.3 Committing a PET computation

Consider a PET computation that has been started as a set of n threads. These threads start executing at their home nodes and some of the threads return successfully. We now describe the commit protocols which can be used to ensure that the results of a single thread are applied to the replicas of the objects invoked by that thread.

The commit protocol takes any one completed thread, and for each object touched by the thread, it tries to apply the updates made by this thread to a majority of the replicas of the object (copying from the updated replica is used for this purpose). This is done atomically, and all the other treads are aborted. Since the PET computation can be completed successfully if a single thread can be committed, the commit can be done sequentially, that is try to commit one thread, and if it fails, repeat the same for another thread. This can also be attempted in parallel; that is, try to commit all the threads, and finally elect one of the successful threads and apply its updates. The sequential commit is simpler and cheaper because the protocol necessary to do it can be executed by the coordinator node. However, it requires that the coordinator remain operational until the PET is committed. A decentralized commit protocol can be used to tolerate failure of the coordinator. We discuss both types of protocols.

To commit a thread, we first make a list of the objects touched by the thread, regardless of whether the object replica was read or updated. This list (and a list of nodes visited by the thread) can be built as the thread traverses various replicas and it can be returned to the home node of the thread when the invocation returns. We then obtain quorums for each of the touched objects [9]. A quorum is obtained if at least a majority of votes are collected and the replica touched by the thread has the latest version number. When there are few failures (the typical case), the number of nodes that respond to a quorum request will be higher than a majority. In the commit protocol, the replica state is updated at all these nodes. This is done to maximize the likelihood that threads execute with the most up-to-date state of an invoked object. Since a thread touches a replica without

checking if it is current, updating as many replicas as possible will increase the probability that replicas touched by the thread are not found to be outdated when the quorum is collected.

We choose majority quorums for both read and update operations to provide maximum fault-tolerance for both types of operations. When operations that only read are more frequent, a smaller read quorum can be used to reduce the cost of implementing them. However, the fault-tolerance for update operations will decrease because a higher write quorum will be required. If a lesser degree of fault-tolerance is acceptable for update operations then a smaller quorum for read operations can be used when most operations are of the type read. Thus, an arbitrary quorum assignment can be used in place of the quorum used in this and the following sections.

After all the quorums are successfully obtained, the updates to replicas touched by the thread are propagated. This is done by copying the updated state of the touched replica to the corresponding state of all replicas that constitute the quorum for the object. Note that we could have obtained a read quorum when the object replica was touched, and a write quorum when the replica is finally updated. We choose to use an optimistic approach and obtain quorums only after the execution of the thread has terminated. Thus, the quorum collection is done only for committing threads and not for all threads of a PET computation.

A node starts quorum collection for an object by sending a message to all nodes that store replicas of the object. This message identifies whether the quorum is being collected for a read or write operation. An operational node responds to such a request by sending a response and it also places a lock on the replica. A read lock is placed when the quorum collection is for a read operation, otherwise a write lock is placed. A node responds only when the appropriate lock can be obtained. For example, if some thread belonging to a different computation already has a write lock on a replica, the node will not respond to quorum requests for threads of other computations until the write lock is released. Since read locks can be granted to any number of threads belonging to the same or different computations, a node can respond to a read quorum request immediately when a write lock does not exist on the replica. Thus, read locks by different threads are compatible. Furthermore, write locks are compatible when they are requested by threads of the same computation. If a write lock is granted to thread $A_{i,j}$ and then a write or read quorum request arrives from $A_{i,k}$ ($A_{i,j}$ and $A_{i,k}$ both belong to the same computation A_i), then also the node can respond to the request. We can do this because only one of the threads will commit successfully. Notice that no lock is placed on a replica when it is touched. A lock is placed only when a quorum request is received.

Let $RT(A_{i,j})$ be the set of replicas which have been invoked by thread $A_{i,j}$. We also define the set of objects, $OT(A_{i,j})$, which are invoked by the thread as follows.

$$OT(A_{i,j}) = \{O_l \mid O_{l,r} \in RT(A_{i,j})\}$$

Each replica $O_{l,r}$ of object O_l has a version number, $VN(O_{l,r})$ associated with it. All version numbers are set to zero when the replicas are created and $VN(O_{l,r})$ is incremented each time a thread that updated $O_{l,r}$ is committed.

Centralized commit protocol. We assume a PET computation A_i is under execution. At some point, a thread $A_{i,j}$ completes by returning to its home node $H(A_{i,j})$ which informs the coordinator node $C(A_i)$ of the completion of the thread. Suppose $A_{i,j}$ is the first thread to notify $C(A_i)$. Now $C(A_i)$ executes the following steps to see if the thread can be committed. The algorithm used is a general version of the 2-phase commit protocol.

- $C(A_i)$ sends a pre-commit message to nodes where replicas in $RT(A_{i,j})$ are stored. On receiving a message, each node creates a stable version of its data segment(s) of the replicas in $RT(A_{i,j})$ and replies with a positive acknowledgement.
- If all nodes that store replicas in $RT(A_{i,j})$ reply with positive acknowledgements, $C(A_i)$ proceeds, else it aborts this thread (as in the pre-commit phase of the 2-phase commit protocol). If no message is received from a node, the coordinator timeouts and assumes a negative acknowledgement (assumes that the node has failed).
- Now, $C(A_i)$ must collect quorums for touched objects. This is done by sending a request to *all* nodes that store replicas of each object touched by the thread. That is, the request is sent to nodes where replicas in $\{O_{l,r} \mid O_l \in OT(A_{i,j})\}$ are stored. The quorum message requests a read lock for object O_l if the thread did not update the object state. Otherwise, a write lock is requested. If a lock cannot be obtained due to a conflict, then the quorum request is kept pending. When the lock is obtained, then the node having replica $O_{l,r}$ replies to this request with the version number $VN(O_{l,r})$.
- The coordinator is successful in getting the quorum if:

$$\forall O_l \in OT(A_{i,j}) \{\mathrm{CARD}(\mathrm{Qset}) > \mathrm{MAJ}(O_l)\}$$

Where CARD denotes the cardinality,

$$\mathrm{Qset} = \{O_{l,r} \mid VN(O_{l,r}) \leq VN(O_{l,t}) \wedge O_{l,t} \in RT(A_{i,j})\}$$

and $\mathrm{MAJ}(O_l) = \mathrm{CARD}(\{O_{l,t}\})/2 + 1$.

That is, for all the objects touched by the thread, at least a majority of the replicas should be in the quorum with appropriate locks placed on them, and the version numbers of these replicas must be less than or equal to the version number of $O_{l,t}$, the replica touched by $A_{i,j}$. Since at least a majority of the replicas of an object and their version numbers are updated when a thread commits, the version numbers of replicas not in the quorum cannot be higher than $VN(O_{l,t})$. Thus, when a quorum is collected successfully, $A_{i,j}$ has executed with the most up-to-date state of the object.

- If the quorum cannot be collected, the thread is aborted and the locks are released by sending a message to the nodes that responded positively to the quorum

request. Otherwise, the coordinator records this on stable storage and for each object in $OT(A_{i,j})$, it asks the nodes that participated in the quorum to commit. Let t denote the replica of O_l that is touched by the thread $A_{i,j}$ of PET A_i which is being committed. That is, if $O_l \in OT(A_{i,j})$ then $O_{l,t} \in RT(A_{i,j})$. The commit is done as follows.

for all $\{O_{l,r} |$ node storing $O_{l,r}$ participated in the quorum$\}$

if $r \neq t$ then the data segments of $O_{l,r}$ are replaced with the data segments of $O_{l,t}$, and the version number of $O_{l,r}$ is set to the new (incremented) version number of $O_{l,t}$.

if $r = t$ then the pre-committed version is made permanent and $VN(O_{l,r})$ is incremented.

The state of a touched replica is made permanent and this state is propagated to the replicas that consitute the quorum. The version number of each updated replica is set to the new version number of the touched replica. The failure of a node that participated in the quorum before the termination of the commit protocol will delay the completion of the commit protocol until the node recovers even if a majority exists without it (the protocol can be easily modified so it can terminate when the state has been propagated to a majority of nodes). Since the locks are recorded in stable storage and the segments of touched replicas are also stored in stable storage at the nodes where the thread executed, the replica state will be propagated when nodes recover from failures.

If $A_{i,j}$ is aborted, any nodes that sent messages are informed so they can release locks. The coordinator waits for another thread to complete and repeats the algorithm. When a thread is committed, the coordinator informs the home nodes of remaining threads, which are then aborted. If all threads get aborted, the PET fails.

In the above algorithm, the decision to commit or to abort a thread is made when the quorum is collected. The locks prevent any of the participants in the quorum from responding to conflicting quorum requests of other computations. If messages get lost, the participants remain blocked and have to check back with the coordinator after timeouts. If the coordinator fails during commit, the home nodes of completed threads may have to wait for the coordinator to become operational again. The decentralized commit protocol avoids this by allowing the home nodes to coordinate between themselves to reach a decision.

Decentralized commit protocol. The decentralized commit protocol does not require that the coordinator remain operational until a thread is committed. The coordinator starts the computation at the home nodes but does not have to participate in the committing of a thread. Each home node receives a list containing all the home nodes of the threads that execute the PET computation. The home node of thread $A_{i,j}$, $H(A_{i,j})$ takes the following actions when $A_{i,j}$ returns successfully.

- $H(A_{i,j})$ sends a pre-commit message to nodes storing all replicas touched by the thread $A_{i,j}$, that is $RT(A_{i,j})$. A node responds to such a message immediately. If $H(A_{i,j})$ does not receive a response from some node then it assumes that the node has failed and the thread is aborted. This decision is recorded by $H(A_{i,j})$.
- When all nodes storing replicas in $RT(A_{i,j})$ respond positively, $H(A_{i,j})$ attempts to obtain a quorum. The quorum is obtained in the same way as before. Since a lock granted to a thread does not conflict with threads of the same PET computation, each replica may participate in the quorum of many threads of a single PET computation.
- If $H(A_{i,j})$ succeeds in obtaining a quorum, it asks nodes that store replicas in $RT(A_{i,j})$ to propagate the updated data segments of the touched replicas to the replicas participating in the quorum. Since each replica may participate in several quorums, it is possible for it to receive several different data segments. It stores all these data segments separately, tagged by the thread identifier. It is not required that the data segments be stored in stable storage.
- After successful state propagation, $H(A_{i,j})$ decides to attempt to commit $A_{i,j}$. At this point, we use an election protocol [8] among all $H(A_{i,k})$ such that $A_{i,k}$ has completed. The elected node is chosen to be the candidate for commit, and it executes a 2-phase commit with the participants in its quorum as the cohorts. The cohorts make the state of the committed thread permanent, and discard any data segments received from other threads.
- If the elected node fails, or is unable to commit because a node in its quorum fails, an election is conducted again and the new winner is chosen from the home nodes of the remaining threads. This is repeated until a thread is committed. If all home nodes fail or are unable to commit their threads, the PET fails.

After successful commit of $A_{i,j}$, $H(A_{i,j})$ informs all the other home nodes, which abort the threads controlled by them.

The decentralized commit protocol uses a greedy algorithm since it attempts to propagate the states of all completed threads before making the commit decision. On the other hand, the centralized commit protocol takes a lazy approach for state propagation since it is done only after a commit decision has been reached. The greedy state propagation requires that a larger number of messages are sent but it commits a thread when the centralized commit protocol cannot do so because of a particular sequence of failures. Several variations in between these two algorithms can be easily derived. Also, many optimizations are possible in the algorithms presented in this section.

We use an optimistic approach and do not lock replicas constituting a quorum set at the time a replica is invoked. Therefore, it is possible that threads executing on behalf of two actions can execute with different replicas of an object and consequently neither of them can collect the required quorum. In this case, both coordinators will abort after a timeout period because they will assume that the nodes that do not respond have failed. Thus, timeout by the coordinators prevents deadlocks.

6 Correctness

The correctness of the PET scheme will follow if it can be shown that the execution of an action is *single-copy* serializable with respect to other actions. Furthermore, the state of the object replicas resulting from the action using PET should also be obtainable when the action is executed by a single thread. Since an action commits when a thread is committed after collecting an appropriate quorum, one-copy serializability follows. We do allow conflicting locks on a replica to be granted simultaneously to threads of the same computation. This does not affect correctness because the commit protocols ensure that a single thread is committed even when quorums are obtained by home nodes of multiple threads.

The permanent state of the replicas is changed only when a thread commits and since a single thread is committed, the result is the same as if the action was executed by a single thread when no communication takes place between the different threads. If two threads do not invoke a single replica then there will be no communication between them. In the case when a common replica is invoked, each thread gets a private copy from the single permanent state of the replica and works with its own copy. Thus, there cannot be any communication between them. The PET scheme can handle non-deterministic computations because the effect is the same as if a single thread executes the action.

7 Performance issues

The system is available to an action if for each object touched by the action, the number of replicas of the object at operational nodes is equal to or greater than the number required for the completion of the action. *Availability*, which is used as a measure of fault-tolerance, is defined as the steady state probability of a system being available to an action. Using the quorum consensus method, an action can commit only if the set of operational nodes that store replicas of an object touched by the action constitute a quorum. Thus, when the quorum is q, the availability of an object to the action is limited by the probability of having at least q replicas at operational nodes. To be able to tolerate maximum number of failures for both read and write operations, the value of q is set to the majority of N, which is the total number of replicas of an object. Thus, we define system availability as the steady state probability of having at least a majority of replicas at operational nodes.

In a system state in which each object invoked by an action is available (at least a majority of its replicas are at operational nodes), the scheme used for executing the action should be able to commit it when problems such as locking conflicts do not arise. In the PET scheme, this may not be possible if all threads of an action encounter failed nodes. Thus, an action is committed with the probability that at least one thread does not invoke a replica stored at a failed node. It can be easily seen that as the number of threads is increased, this probability will increase because each thread tries to execute with a different set of replicas. Since each thread makes remote calls (when necessary) independently of others, the communication cost of executing the action computation will increase with the number of threads. In this section, we develop a simple model to estimate the number of threads that need to be created to achieve a given degree of fault-tolerance. The communication cost of the PET as well as other schemes is considered in Sect. 8.

7.1 Number of threads

If each node in the system has up-to-date information about failures and the operational nodes constitute a quorum for each object to be touched by a thread, a single thread can execute with available replicas and the action can be committed when no transient failures occur (nodes storing replicas touched by the thread do not fail before it is committed). The use of multiple threads by the PET scheme guards against transient failures while not requiring accurate information about failed nodes because each thread potentially executes with a different set of replicas.

We use a simple model to study the relationship between the probability of committing an action and the number of threads. We assume that the steady state probability of a node being operational is p. We also call p the reliability of each node. Let $P_{suc}(\mathscr{X}, N, l, m)$ be the probability of committing an action when each object is replicated at N nodes, l nodes are visited by a thread and m threads execute the PET computation. The parameter $\mathscr{X}$ denotes the scheme used for choosing replicas invoked by a thread which has an impact on the probability of success. The action can be committed only when at least one of the m threads returns successfully (it invokes replicas which are at operational nodes) and quorums are available for objects touched by the threads. To make the analysis feasible, we will compute $P_{suc}(\mathscr{R}, N, l, m)$ where $\mathscr{R}$ is the scheme in which a node storing a replica of the invoked object is chosen randomly when such a replica is not available at the nodes already visited by the thread. Thus, each replica is chosen with the same probability and no information about currently operational nodes is needed. It can be easily seen that $P_{suc}(\mathscr{S}, N, l, m) \geq P_{suc}(\mathscr{R}, N, l, m)$, where $\mathscr{S}$ is the scheme described in Sect. 5.1. Thus, $P_{suc}(\mathscr{R}, N, l, m)$ provides a lower bound for $P_{suc}(\mathscr{S}, N, l, m)$, and typically the number of threads required to get a certain level of fault-tolerance will be smaller than the number derived from the analytical results. We also assume that when a thread touches replicas of two objects which are at different nodes, the sets of nodes storing replicas of each object are disjoint. This implies that such objects are replicated at nodes with independent failure modes. This allows us to assume that the probabilities of successful quorum collection for objects invoked at different nodes are independent. It may not be true for real systems but $P_{suc}(\mathscr{R}, N, l, m)$ will still be a lower bound for $P_{suc}(\mathscr{S}, N, l, m)$ when the assumption is relaxed.

Let l be the number of nodes at which a given thread executes. (In the DSM approach, l is the number of nodes

that supply replicas to the thread.) With the assumptions that objects invoked on different nodes do not have replicas at common nodes and when $N=2k+1$, the probability of success is,

$$P_{suc}(\mathscr{R}, N, l, m) = P^l_{maj} - \sum_{n_1=k+1}^{N} \sum_{n_2=k+1}^{N} \cdots \sum_{n_l=k+1}^{N} Q_{n_l} Q_{n_{l-1}} \cdots Q_{n_1} \cdot \left(1 - \prod_{i=1}^{l} \frac{n_i}{N}\right)^m$$

where

$$P_{maj} = \sum_{n_l=k+1}^{N} \binom{N}{n_l} p^{n_l} (1-p)^{N-n_l}$$

and

$$Q_{n_i} = \binom{N}{n_i} p^{n_i} (1-p)^{N-n_i}$$

A detailed derivation of $P_{suc}(\mathscr{R}, N, l, m)$ can be found in the Appendix.

Table 1 shows P_{suc} for various values of m when $l=1$ (replicas of objects invoked by a thread are available at a single node). To illustrate how P_{suc} depends on the degree of replication, N, and the reliability of each node, p, several combinations of values for these parameters are considered. We see that when $N=3$ and $p=0.9$ (on the average, each node is operational 90% of the time), the probability that a single thread can be used to complete the computation successfully is 0.891. This probability is slightly lower than p because not only the node storing the chosen replica must be operational but a quorum must also be available for successfully completing the computation. It can be seen from the formula for P_{suc} that when $l=1$, the probability of success is bounded by P_{maj} which is the probability that quorum is available. For $N=3$ and $p=0.9$, P_{maj} is 0.972 and it is not necessary to create a large number of threads to achieve a probability of success close to P_{maj}. $P_{suc}>0.96$ even when $p=0.9$ and three threads are used to execute the action. This is true for both $N=3$ and $N=5$. If $p=0.99$, $P_{suc}>0.998$ when three threads are used to execute the computation. Since nodes are highly reliable in practical systems, it is not necessary to have a large degree of replication to ensure that a majority of replicas are available with a very high probability. It can be seen from Table 1 that in such systems, a PET computation can be committed with a probability that is close to the availability of the system with a modest number of threads.

In Table 2 we illustrate the impact of distribution on P_{suc} by considering the case when $l=3$. Thus, each thread executes at 3 different nodes because replicas of the objects invoked by the threads are distributed. There are several reasons why replicas of all objects cannot be stored at a single node. These include limitation on storage space at each node, ownership of objects and the need to keep them at nodes where they are used most often. It can be seen that a higher number of threads are necessary for achieving a given probability of success for a PET computation when l is increased from 1 to 3. Since P_{suc} is the probability when the replica to be used by a thread is chosen randomly, we see that when $p=0.9$ and $N=3$, a computation executed by a single thread will complete successfully only with probability 0.707. This is due to the fact that all three nodes chosen to execute the thread computation must be operational and quorums must be available. In fact, P^l_{maj}, which is the probability that quorums are available for all objects is only 0.918 for this system. The probability of success can be made close to 0.918 by using 5 threads. When each node is highly reliable ($p=0.99$), we see that P_{suc} will be greater than 0.995 when the number of threads in 3 and each thread invokes replicas stored at 3 nodes (in this case, P^l_{maj} is 0.9991). Similar conclusions can be drawn about the system when $N=5$.

Since $P_{suc}(\mathscr{S}, N, l, m) \geq P_{suc}(\mathscr{R}, N, l, m)$, the number of threads required to get a desired probability of success will be no greater (and usually less) when the scheme of Sect. 5.1 is used to choose replicas. For example, in a system with $p=0.9$, $N=3$ and $l=2$, it can be seen from the formula for P_{suc} that 3 threads are needed to ensure that P_{suc} is higher than 0.90. However, if no two threads invoke replicas stored at the same node (the scheme of Sect. 5.1 tries to ensure this), $P_{suc}>0.90$ can be achieved with only 2 threads for the same values of p, N and l. This can be shown easily because if quorums are available, both threads will fail only when one replica of each invoked object is at a failed node (since $l=2$, objects at two distinct nodes are invoked). Since two threads do not invoke the same replicas, each thread

Table 1. P_{suc} for $l=1$

Number of threads (m)	$P_{\mathscr{R}}$			
	$N=3$		$N=5$	
	$p=0.9$	$p=0.99$	$p=0.9$	$p=0.99$
1	0.8910	0.98990	0.89667	0.98999
2	0.9450	0.99643	0.96665	0.99791
3	0.9630	0.99861	0.98415	0.99954
4	0.9690	0.99933	0.98904	0.99988
5	0.9710	0.99958	0.99058	0.99996

Table 2. P_{suc} for $l=3$

Number of threads (m)	$P_{\mathscr{R}}$			
	$N=3$		$N=5$	
	$p=0.9$	$p=0.99$	$p=0.9$	$p=0.99$
1	0.70734	0.97000	0.72093	0.97028
2	0.82831	0.98908	0.87691	0.99338
3	0.87683	0.99559	0.93195	0.99839
4	0.89772	0.99783	0.95417	0.99955
5	0.90742	0.99862	0.96412	0.99984
6	0.91225	0.99891	0.96893	0.99992

must invoke replicas at one operational and one failed node. In this case the probability of success is greater then 0.92 which is much higher than $P_{suc}(\mathscr{R}, 3, 2, 2) = 0.887$.

When the DSM mechanism is used to implement object invocation in a workstation/data server environment, a thread will complete successfully if its home node and the data server nodes that supply replicas used by the thread do not fail. Since the number of data servers is small, each data server will contain replicas of most objects when the degree of replication is high. If all replicas used by a thread are available at a single data server, only two nodes will be touched by a thread when scheme $\mathscr{S}$ is used (home node and a data server) and hence $l = 2$. In such a system, fewer threads will be needed because l will be smaller.

The analysis presented in Sect. 7.1 for the steady state, but in practical systems, there are periods when a large number of failures occur followed by a relatively stable operation of the system. If information exists about the frequency of failures, it can be used to decide the number of threads that need to be created. In many distributed systems, there exists a monitoring service that allows each node to maintain approximate information about the state of other nodes. For example, in *Clouds*, such information is already maintained for reconfiguration purposes. If the node where an action originates believes (based on the information available at it) that there are no or a very small number of failures, it can create a small number of threads. This will reduce the computation and communication load in the system. On the other hand, if there are many node and link failures, a large number of threads should be created to avoid the failed nodes and links. Similarly, the use of a large number of threads is desirable if an action must commit with a very high probability. Thus, not only can the system use the available state information to reduce communication and computation costs, but the PET scheme also provides a trade-off between the degree of fault-tolerance and its cost (computation and communication).

8 Comparison with related work

A large number of algorithms have been proposed to maintain the consistency of replicated data but most of them do not address the problems that can arise when computation is also replicated. In this section, we only describe the ones that address the object and computation replication problem in distributed systems. Other schemes such as multi-version programming (e.g., triple modular redundancy [2]) which have been proposed for handling program faults are also not discussed.

- Replicated programs in CIRCUS

The CIRCUS system [6] addressed the problem of replicated distributed computation. In CIRCUS, when a call is made to a program module that has n replicas, n parallel threads are created and each executes independently with a single replica. It can be easily seen that the caller can receive multiple results, and hence it must *collate* them. Furthermore, a replica can receive a call multiple times when the caller module is itself replicated and hence the callee also needs to collate. In the PET scheme each thread executes as if there is no replication, thus the collation problem does not arise. Furthermore, the CIRCUS scheme requires that the computation defined by each replica be determinisitc because the changes in the state of the replicas are done by executing the operation independently at each replica. This is not required by the PET scheme because it commits a single thread and the effects of the other threads are undone. Thus, it can be used even when computations are non-deterministic.

The CIRCUS scheme has a high communication overhead because the number of messages needed to execute a replicated computation will be $2N^2l$ (N is the degree of replication and there are l nesting levels) because each caller replica will invoke all the callee replicas. On the other hand, in the PET scheme only $2lm$ messages are generated for executing the m threads. For guaranteeing atomicity and consistency, CIRCUS requires that each server communicate with the clients to determine if the results of the computation should be committed. This is done by using the same mechanism except that the servers make replicated calls to the clients and hence the additional communication cost of the commit algorithm is $2N^2l$. In the PET scheme, quorums and a commit protocol are used to ensure atomicity and consistency. The overhead of these algorithms for committing a thread will be less than $4Nl$. Although the PET scheme does need to send the updated state of modified replicas, that is made necessary because we want to handle non-deterministic computations.

- Fault-tolerant distributed objects

The algorithm presented by Birman [5] for implementing fault-tolerant objects does not replicate the computation when a call is made to a replicated object O. The call is executed at a single node called the coordinator. If the execution of the call changes the state of O, the replica at the coordinator is copied to all other nodes where O's replicas exist. If a node fails while it is executing the operation on O, another node where a replica of O exists is chosen to execute the operation. To ensure forward progress, the coordinator sends periodic checkpoints to other replicas and hence a new coordinator can resume a partially completed computation. The failure of the coordinator must be detected and a single other node must be chosen to execute the operation. Algorithms for checkpointing and finding another node that can resume a computation after a failure are not required by the PET scheme but an action needs to be restarted when all threads encounter failed nodes.

The execution of a computation in ISIS in done by a single thread which requires $2l$ messages but there is an overhead of sending checkpoints to all operational nodes that store replicas of an invoked object (other messages are needed to inform nodes when they can discard retained results of calls). The checkpoints can be sent infrequently when there are few failures but in

such a case, the PET scheme can also use fewer threads. In ISIS as with PET, a commit protocol is used when an action completes. Although the overhead of collecting quorums is not there, the concurrency control scheme which is based on the available copies method [4] cannot deal with network partitions. The available copies method can be adapted to the PET scheme but quorums allow it to tolerate partitions.

- Fault-tolerant RPC

The scheme presented by Yap et al. [14] for fault-tolerant RPC avoids the collation problem of CIRCUS by sending a call to a process associated with the primary replica of a module. However, the call must be executed by each replica sequentially before the primary is allowed to return the results. Although the communication cost of executing a computation is $4Nl$, it can only be used when computations are deterministic and network partitions do not occur.

- View-stamp replication

The view-stamp replication algorithm [11] can be used in a system that replicates objects. In this scheme, operation calls are sent to a primary which then propagates the relevant events to other replicas. These updates can be normally sent asynchronously. However, there is a single execution thread and failure of an invoked primary leads to an abort of the action. Other transient failures may be tolerated depending on the new view that results from the reorganization. Similar to the PET scheme, when nodes storing a majority or more of the replicas have failed, actions cannot be committed. Since an action can only be committed when a view contains at least a majority of the replicas, the communication cost associated with propagation and committing the action is similar to the PET scheme.

In all schemes other than the view-stamp and PET, the communication cost of executing the computation depends on N which can only be changed by reconfiguring the system by changing the number of replicas (such a cost may be incurred by the view-stamp algorithm for propagating the state even when the action is aborted). The PET scheme offers the advantage that this cost can be controlled at the time the action is created because it depends on the number of threads and not N. The cost of the quorum and commit protocols and state propagation does depend on N but these are needed with the assumption about the system. Non-deterministic computations require that changes to objects must be copied at other nodes and network partitions require communication with a majority of the nodes. A scheme presented in a recent extension [10] of our earlier work [1] also has some of the advantages offered by the PET scheme. It presents similar algorithms for choosing replicas and a commit protocol.

9 Concluding remarks

An object-based distributed system can use object replication to provide fault-tolerance. If computation of an action is not replicated in such a system, the action will fail when it either invokes a replica at a failed node or a node fails during the execution of the computation. This leads to an abort of the action. The PET scheme reduces the probability of an action being aborted due to failures by replicating the computation. The scheme is simple because each computation thread executes as if there is no replication. Furthermore, the computation and communication costs of executing a fault-tolerant action will be higher only if a greater degree of fault-tolerance is desired.

The implementation of the algorithms required to support the PET scheme will be undertaken after the basic system objects that support the *Clouds* operating system functions are implemented. The *Ra* kernel, which will provide the necessary mechanisms to support the system objects, has already been implemented [3] and debugged. In future, we want to explore the implementation of fault-tolerant services using the PET scheme.

References

1. Ahamad M, Dasgupta P, LeBlanc R, Wilkes T.: Fault-tolerant computing in object based distributed operating systems. In: Proc 6th Symp on Reliability in Distributed Systems, March 1987
2. Avizienis A: The *n*-version approach to fault-tolerant software. IEEE Trans Software Eng 11(12): 1491–1501 (1985)
3. Bernabéu Aubán JM, Hutto PW, Khalidi MYA, Ahamad M, Appelbe WF, Dasgupta P, LeBlanc RJ, Ramachandran U: The architecture of *Ra*: a kernel for *Clouds*. In Proc 22nd Annu Hawaii Int Conf on System Sciences, January 1989
4. Bernstein PA, Goodman N: An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans Database Syst 9(4):596–615 (1984)
5. Birman K, Joseph T, Raeuchle R, El Abbadi A: Implementing fault-tolerant distributed objects. IEEE Trans Software Eng 11(6):502–508 (1985)
6. Cooper E: Replicated distributed programs. In: Proc 10th ACM Symp on Operating Systems Principles, December 1985
7. Dasgupta P, LeBlanc RJ, Appelbee W: The *Clouds* distributed operating system. In: Proc Int Conf on Distributed Systems, June 1988
8. Garcia Molina H: Elections in a distributed computing system. IEEE Trans Comput C-31(1):48–59 (1982)
9. Gifford D: Weighted voting for replicated data. In: Proc 7th Symp on Operating Systems (Pacific Grove, California). ACM, December 1979
10. Ng TP, Shi SSB: Replicated transactions. In: Proc 9th Int Conf on Distributed Computing Systems, pp 474–480. IEEE, June 1989
11. Oki B, Liskov B: Viewstamped replication: a general primary copy method to support highly-available distributed systems. In: Proc 7th Symp on Principles of Distributed Computing, August 1988
12. Ramachandran U, Ahamad M, Khalidi MY: Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. In: Proc Int Conf on Parallel Processing, August 1989
13. Stonebreaker M: Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Trans Software Eng 5(3):188–194 (1979)
14. Yap KS, Jalote P, Tripathi S: Fault tolerant remote procedure calls. In: 8th Int Conf on Distributed Computing, June 1988

Appendix

We now present the derivation of P_{suc}. As in Sect. 7, we assume that all threads invoke replicas stored at l distinct nodes. Furthermore, when replicas of two objects are invoked at different nodes, the sets of nodes that store the replicas of the two objects are disjoint. This will allow us to assume that when a thread needs to invoke a replica at a node that has not been visited by it, the probability that it will choose an operational node is independent of the nodes that have been visited by the thread. We assume $N=2k+1$ and the steady state probability of a node being operational to be p.

We define the state of the system in which a PET computation executes in terms of the number of operational nodes that store replicas of objects touched by the threads of the computation. In particular, when threads execute at l distinct nodes, let $\underline{S}=(n_1, n_2, \ldots, n_l)$ be a given system state in which $n_i (0 \le n_i \le N)$ of the N replicas of the i^{th} group of objects are at operational nodes. This group of objects is defined by the replicas touched at the i^{th} node. Let Δ be the set of possible system states. As defined in Sect. 7, the probability of successful completion is $P_{suc}(\mathscr{R}, N, l, m)$, where m is the number of threads and l and N are as defined above. Since a computation can be committed only if at least one of the thread returns successfully and quorums are available for all touched objects (e.g., majority of replicas are at operational nodes), we can express $P_{suc}(\mathscr{R}, N, l, m)$ as follows:

$$P_{suc}(\mathscr{R}, N, l, m) = \sum_{\underline{S} \in \Delta} Prob.[\underline{S}]\ Prob.[\text{at least one of } m \text{ threads succeeds in state } \underline{S} \text{ and quorums are available for all touched objects in state } \underline{S}]$$

We compute $1-P_{suc}(\mathscr{R}, N, l, m)$ and denote it by $P_{fail}(\mathscr{R}, N, l, m)$. Thus, $P_{fail}(\mathscr{R}, N, l, m)$ is the probability that the computation is not successful either because quorum was not available for some touched object or all threads encountered failed nodes. To compute P_{fail}, we partition the states in the set Δ. Let Δ^+ be the subset of states such that each state in it has at least a majority of replicas of each touched object at operational nodes. Therefore, if $\underline{S}=(n_1, n_2, \ldots, n_l) \in \Delta^+$ then $n_i > k$ for $1 \le i \le l$. Let $\Delta^- = \Delta - \Delta^+$, which is the subset of states in which quorums are not available for one or more objects. Since a computation must fail when a quorum is not available for some object that it touches, all states in Δ^- lead to failure. Furthermore, the computation can fail even in a state that belongs to Δ^+ because all threads may encounter failed nodes. Thus, we can write,

$$P_{fail}(\mathscr{R}, N, l, m) = \sum_{\underline{S} \in \Delta^-} Prob.[\underline{S}] + \sum_{\underline{S} \in \Delta^+} Prob.[\underline{S}]\ Prob.[\text{all } m \text{ threads fail in state } \underline{S}]$$

We first compute $\sum_{\underline{S} \in \Delta^-} Prob.[\underline{S}]$.

$$\sum_{\underline{S} \in \Delta^-} Prob.[\underline{S}] = 1 - \sum_{\underline{S} \in \Delta^+} Prob.[\underline{S}] = 1 - \sum_{n_1=k+1}^{N} \sum_{n_2=k+1}^{N} \cdots \sum_{n_l=k+1}^{N} Q_{n_l} Q_{n_{l-1}} \cdots Q_{n_1}$$

where

$$Q_{n_i} = \binom{2k+1}{n_i} p^{n_i} (1-p)^{2k+1-n_i}$$

Since Q_{n_i} does not depend on n_j when $j \neq i$, the above can be simplified to,

$$\sum_{\underline{S} \in \Delta^-} Prob.[\underline{S}] = 1 - P_{maj}^{l}$$

where

$$P_{maj} = \sum_{n=k+1}^{N} \binom{2k+1}{n} p^n (1-p)^{N-n}$$

is the probability that at least a majority of replicas of an object are at operational nodes.

To complete the derivation of $P_{fail}(\mathscr{R}, N, l, m)$, we now derive the probability that the computation fails even when quorums are available for all touched objects. In state $\underline{S}=(n_1, n_2, \ldots, n_l)$, the probability that a thread executes at operational nodes is $\frac{n_1}{N} \cdot \frac{n_2}{N} \cdots \frac{n_l}{N}$. Thus a thread fails with probability $\left(1 - \prod_{i=1}^{l} \frac{n_i}{N}\right)$. Since threads choose replicas independent of each other, the probability that all m threads fail is simply $\left(1 - \prod_{i=1}^{l} \frac{n_i}{N}\right)^m$. Therefore,

$$\sum_{\underline{S} \in \Delta^+} Prob.[\underline{S}]\ Prob.[\text{all } m \text{ threads fail in state } \underline{S}] = \sum_{n_1=k+1}^{N} \sum_{n_2=k+1}^{N} \cdots \sum_{n_l=k+1}^{N} Q_{n_l} Q_{n_{l-1}} \cdots Q_{n_1} \left(1 - \prod_{i=1}^{l} \frac{n_i}{N}\right)^m$$

Therefore,

$$P_{suc}(\mathscr{R}, N, l, m) = 1 - P_{fail}(\mathscr{R}, N, l, m) = 1 - (1 - P_{maj}^{l}) - \sum_{n_1=k+1}^{N} \sum_{n_2=k+1}^{N} \cdots \sum_{n_l=k+1}^{N} Q_{n_l} Q_{n_{l-1}} \cdots Q_{n_1} \left(1 - \prod_{i=1}^{l} \frac{n_i}{N}\right)^m = P_{maj}^{l} - \sum_{n_1=k+1}^{N} \sum_{n_2=k+1}^{N} \cdots \sum_{n_l=k+1}^{N} Q_{n_l} Q_{n_{l-1}} \cdots Q_{n_1} \left(1 - \prod_{i=1}^{l} \frac{n_i}{N}\right)^m$$