Voting strategies for Thomas's Majority Consensus Method
by William Jay Hutchison

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
This paper investigates the performance of several strategies for ordering votes during update synchronization for transactions accessing replicated or multiple copy databases managed by Thomas's Quorum Consensus Method. Strategies for fixed order voting, random order voting, and shortest paths voting are presented and evaluated. Two methods for refreshing rejected transactions are evaluated and compared. A simulation of a distributed replicated database system residing on a network with update transaction access managed by Thomas's Method is described. A suite of workloads is developed to exercise the simulation facilitating the evaluation of the refresh methods and vote order strategies. The results from the simulation are presented. The average vote probe count,, transaction response time, and transaction throughput achieved by each strategy are examined.

Fixed order voting is shown to have the property of fast conflict detection and resolution. No advantage to either refresh method is found.

VOTING STRATEGIES FOR THOMAS'S MAJORITY CONSENSUS METHOD

by

William Jay Hutchison

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 1992

APPROVAL

of a thesis submitted by

William Jay Hutchison

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

12/18/92
Date

Chairperson, Graduate Committee

Approved for the Major Department

December 18, 1992
Date

Head, Major Department

Approved for the College of Graduate Studies

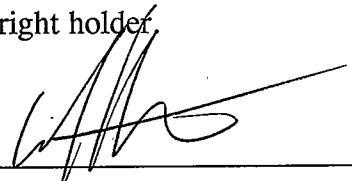
12/21/92
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under the rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature 

Date December 18, 1992

ACKNOWLEDGMENT

The work reported in this paper was supported by the Hewlett-Packard Company.

TABLE OF CONTENTS

TABLE OF CONTENTS—Continued

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

This paper investigates the performance of several strategies for ordering votes during update synchronization for transactions accessing replicated or multiple copy databases managed by Thomas's Quorum Consensus Method. Strategies for fixed order voting, random order voting, and shortest paths voting are presented and evaluated. Two methods for refreshing rejected transactions are evaluated and compared. A simulation of a distributed replicated database system residing on a network with update transaction access managed by Thomas's Method is described. A suite of workloads is developed to exercise the simulation facilitating the evaluation of the refresh methods and vote order strategies. The results from the simulation are presented. The average vote probe count, transaction response time, and transaction throughput achieved by each strategy are examined.

Fixed order voting is shown to have the property of fast conflict detection and resolution. No advantage to either refresh method is found.

CHAPTER 1

INTRODUCTION

As distributed systems gain acceptance in universities and businesses, replicated file and database systems are becoming important. The widespread use of networks to interconnect computers naturally leads to the desire to share information through access to files and databases located throughout the network. Many techniques have been developed to provide this type of access. Among them is the use of duplicates or replicates of a database spread through the network. Distributed replication allows continued access to the database in case of a database server host failure. Improvements in access performance are also provided by distributing the access workload among multiple computers and by shortening the communications distance between any user and a copy of the database.

Access to the database replicates must be managed in such a way as to guarantee consistency among the copies while allowing multiple users to access the database. This is generally achieved by applying some scheme to enforce serial equivalence among concurrent read and update database transactions. Serial equivalence requires that the state of the database after any two concurrent transactions complete be equivalent to the state the database would achieve if the two transactions were entirely serial. If two transactions conflict, serial equivalence cannot be achieved and one of the transactions must be aborted. Serial equivalence does not specify in which order the transactions would execute, only that the database be left in one of the two possible states after both transactions complete.

The two primary methods of managing access to replicated databases are master/slave and distributed access. Master/slave systems have a single primary copy of the database with all changes made to the primary and then promulgated out to the replicates. While quite efficient in normal operation, the loss of a database master server cripples the ability to update the database. Distributed methods are more robust because there is no single point of control. Distributed management prevents a single server failure from halting database update operations.

Distributed methods usually require an updating process to gain permission from each member of some subset or quorum of the copies to update the database. Permission is acquired by having the copies vote to allow the updating process to continue. If the updating process cannot assemble the required quorum of affirmative votes, permission is denied and the process must retry later. This voting technique was first described by R. H. Thomas in [Thomas, 1979][1]. The technique is generally known as Thomas's Majority Consensus Method. In order to update a replicated database, an application process must gather a simple majority of assenting votes from the database copies. The method does not describe any particular strategy for organizing or ordering the voting to quickly resolve update permission.

The majority of published research into distributed voting methods for managing access to replicated files and databases has concentrated on reducing the size of a quorum needed to gain permission to update a file or a database. The various papers on the topic have described organizations of copies that result in significantly smaller voting sets while maintaining most of the desirable attributes of a replicated database system including consistency, reliability and performance. Some of this research will be described in Chapter 2.

No research has been found which explores ordering voting to reduce the time and number of operations required to achieve a quorum. As noted above, Thomas did not

specify any particular order or strategy for gathering votes. While later methods may provide some improvement in quorum sizes, Thomas's original approach is elegant and straightforward and has been a mainstay of the literature regarding replicated file and database management. Thus the research reported here will concentrate on enhancing Thomas's Method by exploring several strategies for ordering the voting to improve the performance of update synchronization for a replicated database system.

The need for concurrent access control and several methods from the literature will be discussed. Thomas's Majority Consensus Method will be described in some detail. The system performance aspects of voting organization in the operation of this method will be singled out for further investigation. To quantify the performance effect of several possible voting strategies, a simulation of Thomas's Method will be developed.

The results of the simulation will be analyzed and conclusions about the applicability of the strategies will be drawn.

CHAPTER 2

REPLICATED DATABASES AND MUTUAL EXCLUSION

Replicated Databases

While concurrent access management methods have general application to data set and item locking in distributed databases, this paper will focus on transaction management for replicated databases. A database system with copies of each data set on separate machines is generally known as a *distributed replicated database service*. A number of independent computers collaborate to maintain consistent copies of critical or frequently accessed databases presenting a fast, reliable and cohesive database system service to client processes. In most cases, each computer bears partial responsibility for controlling access to the set of duplicates.

The objectives of replicating databases and files around a network are improved reliability and performance. Having multiple copies of an important database located on independent but cooperating computers allows continued access in the event of one or more system or network component failures. As long as a client is able to reach a server hosting a current copy of the target data set, processing may continue.

Where a database is accessed by processes running on many distributed systems and is frequently read but is only occasionally updated, multiple copies can be used to reduce communications overhead and read response time. Locating a copy on or topologically near a client process system reduces the number of network transactions required and shortens the average distance each transaction must travel. In addition, the client request load is concurrently shared by the file servers, improving file system

throughput. However, other methods of improving reliability and performance may provide better results where a database is frequently written. Sophisticated transaction concurrency control and serialization techniques such as distributed two phase commit protocols may be required.

Mutual Exclusion

Mutual exclusion is the requirement that access to a shared resource be limited to a single process at a time. Mutual exclusion is necessary for a shared resource if sharing processes will interfere with each other's use of the resource. Mutual exclusion is used in operating systems and in other applications that manage shared resources. A shared resource may be a piece of physical hardware such as a printer where interleaved access would result in confused output or a logical data structure such as a table or linked list where several contiguous operations must made without interference to maintain a consistent and correct structure from access to access. If two or more processes are attempting to simultaneously access a resource for which mutual exclusion is required, the process' access must be serialized. One process must be allowed to complete its activity with the resource before another is allowed to begin. Among the low level locking facilities often used to implement mutual exclusion for more than two processes are test-and-set locks, semaphores[2], and monitors.

A test-and-set lock is bit in memory that is accessed using a hardware test-and-set operation. A process attempting access to a lock controlled resource does a test-and-set on the lock. This operation notes the value of the bit, sets it and returns the original value. If the original value was reset, the executing process did the set and is allowed to continue. If the original value was set, the process is denied access and must try again. Once access is gained, the process uses the resource and resets the lock bit

allowing other processes access to the resource. This is a very low-level operation usually implemented in hardware.

A semaphore is a protected variable that can only be accessed and altered by two atomic software operations, P and V, named for Dutch railroad traffic controls, and an initialization routine. A process needing access to a resource controlled by semaphore executes P. P tests the value of the semaphore. If the value is positive, the process decrements it, gaining access to the controlled resource. If the value is not positive, the process enters a wait queue and is suspended. Once a process has completed its controlled activity with the resource, it increments the semaphore value and, if there are any, removes a suspended process from the wait queue. The revived process should now be able to decrement the semaphore and gain control of the shared resource. Semaphores in one form or another are the usual basis for most software locks.

A monitor is an abstract data type often implemented in concurrent programming languages and system libraries. A monitor is similar in operation to a semaphore but consists of a monolithic program object. A process attempting to use a resource does not implement the access and release operations as with a semaphore. Instead, the process uses a system or application specified monitor to arbitrate access and to manage the process wait queues.

Any of these three increasingly abstract facilities can be used as the building blocks for a high level local lock manager for distributed systems. All that is needed is a way to make the managers available to remote cooperating processes.

Databases are sets of data elements that can be read and written by processes. Two or more processes can safely read all or part of a database simultaneously because each process sees an unchanging, consistent set of elements. However if one or more processes are updating a database, some kind of access control is required to assure

serial equivalency. Two updating processes may not interleave their changes such that the database is left in an inconsistent state. A writing process should not be changing any data being read by other processes as the reading processes may not see a consistent set of data elements. Some elements may be stale and some may be new. Thus some type of mutual exclusion or transaction management is necessary to manage access to databases.

The same access conflicts that occur within a database also occur with a set of replicated databases. If processes are writing to a replicated database, the changes to all the copies must be managed in such a way as to guarantee that all readers see current and consistent database contents and to guarantee that multiple updates do not interfere with each other.

Distributed Transaction Management

The two general approaches to organizing transaction management for replicated databases are *master/slave strategies* and *distributed access control.*

Master/Slave Strategies

Master/slave strategies, such as that used by the Sun Network Information System also known as Sun Yellow Pages Service (YAP)[3], have one primary server and multiple secondary servers for each replicated file or data set. The primary server is responsible for all write access and update transaction serialization. Initial versions of a file or data set and subsequent modifications are propagated by the primary server to the secondary servers. Database service clients may read data elements from any copy but must direct all updates to the master copy. While this method has the advantage of being straightforward with a single point of update control, there are several disadvantages. A reliance on a master copy for update serialization allows a single

failure to preclude further changes. Modifications to the master copy are usually made without locking slave copies. Updates are distributed to slave copies after the master update is complete. Clients may read a stale slave copy until change notification is complete. Master/slave strategies are most appropriate for stable or slowly changing files and databases.

Distributed Access Control

Distributed access control is a more general approach in which there is no single master copy. Access arbitration generally involves several copies. While other mechanisms can be used to manage concurrent access and data set suite consistency, distributed access control allows processing to continue with the loss of some number of copies.

Following are some examples of distributed algorithms for managing access to a set of replicated files and databases.

- Majority Consensus (MC) [Thomas, 1979].
- Quorum Consensus (QC) [Gifford, 1979][4].
- Hierarchical Quorum Consensus (HQC) [Kumar, 1991][5].
- $\sqrt{N}$ [Maekawa, 1985][6] .
- Tree [Agrawal, 1989][7].

All of these techniques involve gaining access permission from each member of some subset of the database suite before reading or updating any copy of a data set. Mutual exclusion is achieved by guaranteeing that any two properly selected subsets will contain at least one member in common and that all members give permission only if the requested access will not interfere with any other active access to that member's data sets. Thus no two conflicting processes will each succeed in simultaneously gaining permission from a complete subset. The various techniques mentioned above

focus on organizing subset selection to reduce the size of a subset while guaranteeing at least one copy intersection between any two allowed subsets.

A database update transaction consists of a set of base elements, data items read from the current database from which the updates are computed, and a set of update elements, the data items to be changed.

Thomas's Majority Consensus Method

Majority consensus (MC) is the simplest and oldest of the algorithms discussed in this paper and is a mainstay of the literature. The concept is fairly straightforward. Each copy of a database in a database suite is given a vote on whether a process may have access to the suite or not. A process attempting to update the database suite must assemble at least a simple majority of copy votes agreeing to the action. Because a majority includes more than half of the files in the suite, every majority will include at least one common copy. The intersecting copy will arbitrate access to the database suite. An update transaction receives a OK vote only if its base elements are current and if the requested updates will not interfere with any other currently active transaction. The algorithm will be further detailed in Chapter 3.

For n copies and any two majorities m_1 and m_2:

$$m_1 + m_2 > n \qquad 2m > n \qquad m \geq \left\lceil \frac{n+1}{2} \right\rceil$$

Figure 1 depicts an example with 5 copies and two processes named *a* and *b*. If process *a*'s transaction has achieved a majority and is accessing the database suite, process *b*'s transaction will not be able to assemble the necessary number of OK votes if the two transactions conflict.

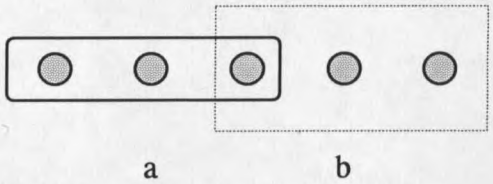


Figure 1. MC example of intersecting simple majorities.

When a majority is achieved, a reading process must inspect the version numbers of all members of the majority and read from the most recent copy. An accepted update must eventually be applied to all members of the database suite. These two requirements guarantee that a reader always gets the most current version of the file and that the database converges to a consistent and up to date state.

The failure or loss of a number of copies up to one less than a majority does not affect the operation of the algorithm except that processes may be delayed while waiting for a response from a failed copy. If the number of available copies drops below a majority, the entire distributed system must redefine the size of a majority.

The main problem with the MC algorithm is that a process must communicate with over half of the copies to establish a majority. A simple extension to Majority Consensus addresses this issue and is discussed in the next section.

Gifford's Quorum Consensus Method

The Majority Consensus algorithm can be easily improved by allocating different copies different numbers of votes. This algorithm is known as Quorum Consensus (QC) and is generally applied to suites of replicated files. As long as the vote allocation is consistent, mutual exclusion can be maintained. Each copy of a file in a suite is given a number of votes. In addition, the number of votes required to read the file is set to a count of votes called the read quorum, q_r. The number of votes required to write the file is set to a potentially different count called the write quorum q_w, The

two quorum values need not be the simple majority, but can be adjusted to reflect probable read/write access patterns as long as both the sum $q_r + q_w$ and the product $2q_w$ guarantee at least one overlapping copy. The intersecting copy will arbitrate write access to the database suite.

$$\text{For } n \text{ total votes:} \quad q_r + q_w > n \qquad 2q_w > n \qquad q_w \geq \left\lceil \frac{n+1}{2} \right\rceil$$

If the access workload is dominated by reads with the occasional write, the read and write quorum size can be adjusted to allow multiple reads. Figure 2 shows example read and write quorums with 5 total votes.

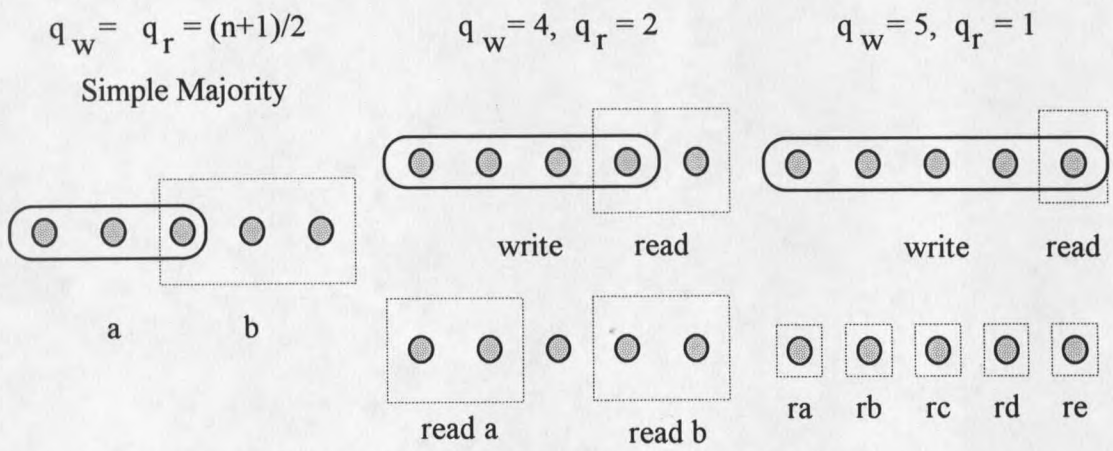


Figure 2. QC examples of adjusting quorum size to favor reading.

The number of votes assigned to a particular copy can be adjusted to reflect the importance of that copy to the file suite. For instance, a particularly fast or centrally located copy may be given more votes to increase the likelihood it participates in a quorum. A copy that is frequently backed up may be favored so that a recent version of the file is usually saved. The vote assignment and quorum sizes must be prearranged and known to all components of the distributed system and all of the accessing processes.