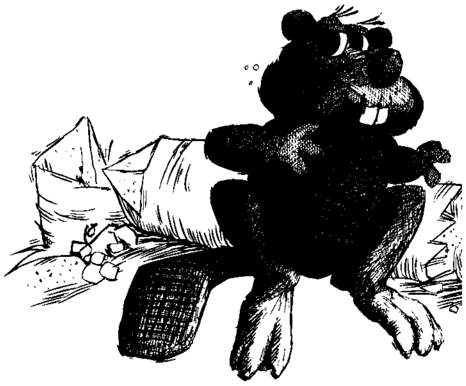

Minimizing Replication Costs with Leaders, Location Information and Optimism

P. Triantafillou
D.J. Taylor

CS-89-52

November, 1989

Minimizing Replication Costs with Leaders, Location Information and Optimism

Peter Triantafillou, David Taylor

Department of Computer Science

University of Waterloo

Abstract

This paper presents a new replication method. Our major goal was to achieve performance similar to systems that do not employ replication and, at the same time, enjoy the availability benefits that result from replication. To reach our goal two mechanisms have been developed: 1) a priority-based preemptive, optimistic concurrency control algorithm and 2) an extended location service, which is incorporated into the transaction processing facility. The method achieves the above performance goal, exhibits good availability characteristics, satisfies the one-copy serializability correctness criterion, is easily proved correct, and is easy to implement.

1 Introduction

Throughout this paper the concept of transactions (atomic actions) is extensively used. Briefly, a transaction consists of a set of primitive operations and is characterized by the properties of *recoverability* and *serializability*. Different researchers assign different properties to transactions (see [Taylor86] for a related discussion) but, in essence, all these properties amount to recoverability and serializability. Recoverability ensures that either all or none of the operations that are included in the transaction have taken effect. Serializability ensures that any execution of concurrent transactions is equivalent to some serial execution.

A distributed system is a collection of computers (nodes) which are interconnected by communication links. These nodes store data and process requests to access their data. Distributed systems are subject to node and communication-link failures. A node failure makes data stored at a failed node inaccessible for the duration of the failure. Similarly, a link failure may cause a node and thus the data contained in it to become inaccessible from some other nodes of the system. The primary motivation for introducing replication in a distributed system is to increase availability.

In distributed systems concurrent access to shared data is common. When one concurrent access is a write access then the system must enforce some concurrency rules in order to protect the internal consistency [Thomas79] of the data. In *one-copy distributed systems*, *serializability* [Papadimitriou79] is widely accepted as the correctness criterion. In *multi-copy* (replicated) systems there exists the additional requirement that the functional behaviour of a replicated object must be equivalent to the behaviour it would exhibit if it were not replicated. Hence, the correctness criterion for systems with replicated data is *one-copy serializability* [Bernstein83]. Informally, one-copy serializability states that any concurrent execution, of actions accessing a replicated object, that is allowed by a replication method must have an effect equivalent to some serial execution of these actions over a non-replicated object.

It is conceptually convenient to view multi-copy objects as a single *logical* object. Operations on a replicated object, although they may be directed to one or more *physical* copies, can be conceptualized as being applied to the corresponding logical object. It is exactly this abstraction of the logical objects that *replication control methods* attempt to support. Note that by ensuring that the functional behaviour of replicated objects is equivalent to that of single-copy data and by using any one of the concurrency control algorithms that are already available [Bernstein81] we can effectively guarantee correctness. From the above discussion it should be apparent that in order to achieve correct behaviour a distributed system with replicated data must have mechanisms to provide the following:

1) Concurrency control in order to preserve the internal consistency of the data, and
2) Replication control to ensure that the functional behaviour of multi-copy objects is equivalent to that of single-copy objects (*one-copy equivalence*).

There are, however, replication methods that do not attempt to ensure one-copy serializability. They allow replicated objects to have mutually inconsistent copies [Oppen81, Demers87, Shroeder84]. In this work we only consider methods that ensure one-copy serializability.

The balance of the paper is organized as follows. Section 2 briefly discusses related work. Section 3 presents the two mechanisms that allow our method to have the desired performance characteristics and presents the method itself. Section 4 reviews the serializability theory for replicated databases and proves the method correct. Section 5 compares the performance and availability characteristics of our method to those of well-known methods. Section 6 outlines the contributions made by the new replication method and contains concluding remarks.

2 Related Work

Quorum-based methods are, probably, the most well known methods. *Quorum consensus* [Gifford79] was the original work introducing the quorum concept and presenting a quorum-based replication method. A quorum is simply a collection containing a specified number of copies of a replicated object. A read operation is performed by first accessing a read quorum, containing r copies. Subsequently, the copy with the highest version number is read. A write operation is performed by writing a write quorum, containing w copies. One-copy serializability is satisfied provided read and write quorums always intersect, thus r and w must be chosen so that $r+w>n$, where n is the total number of copies. The designer gains added flexibility by manipulating r and w and thus the system can be fine-tuned to support query-intensive applications at the expense of update-intensive applications and vice versa. Note, however, that for each operation significantly more work is required for replicated objects, compared to the work required to execute operations on non-replicated objects.

The basic quorum consensus method was extended by Herlihy in several ways. *General quorum consensus* [Herlihy86] introduced the concepts of initial and final quorums and locks, with new and less constraining conflict rules. In this way, greater concurrency can be attained and the performance advantages of this are obvious. In [Herlihy87a] a new method is presented that allows the dynamic adjustment of quorums so that availability can be maximized as a response to network partitions. In [Herlihy87b] two new methods are presented (*consensus locking*, which has good availability characteristics, and *consensus scheduling*, which has good concurrency characteristics) and it is shown that concurrency and availability cannot be maximized simultaneously. It should be pointed out, however, that for each operation within a transaction an initial and a final quorum must be contacted and these additional two rounds of message exchange increase the transaction execution cost significantly.

El Abbadi and Toueg [Abbadi86, Abbadi89] developed a variant of quorum consensus which is based on the concepts of read and write accessibility thresholds, A_r and A_w. As before, reading (writing) requires physically accessing a read (write) quorum of copies. In addition, however, in order for a read (write) operation to be performed, there must exist at least A_r (A_w) accessible copies. The quorum and accessibility threshold values may be defined in a way that gives the designer added flexibility. With the introduction of accessibility thresholds fewer physical operations are needed, since it is never necessary to access more than a single copy during reading. The authors introduce the concept of a *view* that makes the method more robust so that it can tolerate partitioning. The view of a site s is an approximation of the set of sites with which s can communicate. As failures occur, a view-change protocol is invoked and the read and write quorums can be defined with respect to a particular view. This method is referred to as the *virtual partitions* method.

Oki and Liskov [Oki88a, Oki88b] have recently presented a replication method, *viewstamped replication*, which has performance characteristics similar to those of non-replicated systems. It is based on the primary-site paradigm for replication. Operations are executed only at the primary site and results are returned immediately to the client. The other replicas are informed of state changes in the background. During the execution of the two-phase commit protocol, each primary that performed an operation ensures that enough replicas (a majority) know of the effects of that operation before agreeing to prepare. Stable storage is avoided and, instead, the replicas are used to contain the information that would, in non-replicated systems, be written in stable storage. The claim is made that the cost of writing to stable storage is similar to the cost incurred during a round of message exchange between a majority of replicas. A view-change protocol is also provided to handle problems that arise when failures occur.

The *available copies* method [Bernstein83] requires reading from any available copy and writing to all available copies. Special system transactions execute upon the detection of failures to update the information that sites keep regarding the availability of copies. A variant of this method is employed in ISIS [Birman85]. The method is built on top of special broadcasting protocols to ensure that replica coordination occurs in the proper order. Both methods, however, do not guarantee correctness in the presence of network partitions.

The *true-copy token* method [Minoura82] distinguishes between the copies of an object. True copies always have the current state and updates can be performed only on true copies. True copies can be moved from one site to another so that updates can be performed from all sites. Different partitions may update different objects. When the network partition is eliminated the mutually inconsistent states are reconciled by special operations and tools. The method tolerates node failures and network partitioning but the availability of the operations is limited by the number of true copies.

Of the above replication methods, only viewstamped replication exhibits performance characteristics similar to non-replicated systems. This is made possible by employing the primary-site paradigm for replication. However, the method must account for the possibility of failures of the primary node and network failures that may make the primary inaccessible. For this reason, a view-change protocol is developed so that a new primary can be appropriately selected and the effects of committed transactions will survive the failures. This makes the method more complex and difficult to implement. It also limits the availability of the service since replicas remain inactive during the execution of the view-change protocol. Furthermore, viewstamped replication suffers from the well-known disadvantages associated with methods based on primary sites. Most notably, primaries may become performance bottlenecks.

3 Optimistic Location-Based Replication

Our goal was to develop a method that has performance similar to that of non-replicated systems and, at the same time, avoids the disadvantages of the viewstamped replication method. The source of these disadvantages lies in the choice of the primary-site paradigm for replication. However, there are two major benefits of designating one of the replicas as a primary replica. First, it allows transaction operations to execute at only one site (with replica coordination taking place in the background). This results in correct behaviour since the primary always stores the up-to-date information about the replicated objects that it handles. Secondly, it eliminates the need for an expensive transaction synchronization mechanism; replicated lock acquisition to avoid serializability and deadlock problems is not needed. Therefore, we focused our attention on developing

1) A mechanism that allows our method to access only a single replica and still obtain the most recent state for the referenced object, and

2) An inexpensive transaction synchronization mechanism that avoids the need for replicated lock acquisition.

Before we proceed with the description of these two mechanisms, we first present the system model, define some terminology and discuss transaction processing in the model.

3.1 The System Model

A distributed system consists of many sites interconnected by a communication network. We assume that information stored in it is replicated at many sites. As a result, replicated data objects are managed by a group of replica sites. Access to a replicated object can only be provided by the group of replicas that manage it. Thus, using the client-server model we say that the managing replicas provide a service and can view each replica group as a replicated server.

Computations in the distributed system execute as transactions. Each transaction consists of a number of operations, which are classified as read or write depending on the effect the operation has on the object. The site at which a transaction originates is responsible for determining and communicating with the appropriate replica group managing a referenced object and for requesting that the transaction operation be performed on the object. The originating site is referred to as the client and during the execution of a transaction the client will communicate with a number of replicated servers.

The client, before starting to process transaction operations, typically must obtain location information that will inform it of the appropriate replica group to be accessed for each object in the transaction. This information is provided by a location service. Location servers maintain the *location tables* which contain the desired information. This service should naturally be highly available since its availability constrains the availability of transaction operations. The desired availability, in turn, is achieved through replication.

When all transaction operations have been executed and the client has received the results it will initiate the two-phase commit (2PC) protocol. The client will be the coordinator of the 2PC protocol. Transaction operations acquire locks at different sites throughout the system. These locks are held until the transaction is either committed or aborted.

The replicas within a replicated server (eventually) record the effects of transaction operations in a local *event history log*. The use of stable storage is avoided because it is unnecessary: the effects of committed transactions will be reflected in the replicated event history logs and thus reliability can be achieved.

3.2 Location-Based Replication

The essence of our first mechanism lies in the extension of the basic location service, and in the incorporation of this service into the transaction processing facility. For this reason we refer to our method as the *location-based* method. Before we discuss our extensions to this service let us first review its traditional responsibilities.

In a non-replicated distributed system, a client that is about to execute a transaction interacts closely with the location service. For every object, accessed by one of the operations in the transaction, the client queries the location server in order to determine which site in the system stores this object. Subsequently, the operation is forwarded to the storing site where it is executed and results are returned to the client. In a replicated distributed system, the location server may provide the client with a list of sites any (or all) of which are capable of performing an indicated operation. For example, in the viewstamped replication method the location server provides the client with information regarding the group of replicas that control access to a replicated object. It is left to the client to consult enough of these replicas in order to determine the primary site.

In location-based replication, the client interacts with the location server to obtain a list of (one or more) sites that contain an *up-to-date copy* of each replicated object referenced in the transaction. Having obtained this list, the client proceeds with the forwarding of the operations to the appropriate replicas, where they will be executed.

Note, however, that for availability and flexibility reasons we cannot require that there be a fixed set of replicas for an object that will store the most recent information. Therefore, as operations execute there will be a changing set of replicas that have the most recent information about replicated objects. Thus, we must allow for updating the location tables so that the information provided by the location server to the client is always correct. We satisfy the latter requirement by modifying the traditional two-phase commit protocol. In our replication method the client coordinates the execution of 2PC. The method guarantees that at commit time the client knows, for each updated object, which replicas where involved in the execution of the update. Thus, the client knows which replicas are currently aware of the new object state. Hence, along with the commit messages, the client submits a request to the location server so that it can appropriately update the relevant portion of the location table. In addition, we further modify the protocol so that the transaction is not considered committed until all of the replica participants have replied with a "committed" message *and* the location server has replied with an "update o.k." message indicating that the location tables have been appropriately updated.

More precisely, the interface of the location server is defined by the following two operations:

- **server?**(list-of-objects) **returns** list-of(object, list-of-replicas) ;
- **update-table**(list-of(object, list-of-replicas)) **returns** ack ;

The "server?" function is simply the query submitted by the client in order to find out the set of replicas that store an up-to-date copy of the objects involved in the transaction. "list-of-objects" is the list of the objects involved in the transaction. "list-of(object, list-of-replicas)" is a list of pairs. The first component of each pair is one of these objects and the second component is a list of the replicas that store the current state for this object.

The transaction submitting the "server?" query will acquire a read lock at the location server replica where the query is served.

The "update-table" function represents the command submitted by the client during the commit phase of the 2PC protocol that instructs the location server to update the relevant portions of the location table with the information included in the argument. As mentioned earlier, the "update-table" operation involves the cooperation of the location server replicas which collectively perform the update. The leader location server replica is the one that received the "update-table" command from the client. The leader will broadcast the command to the other replicas. Before each server replica performs the update, a write lock is first acquired locally and then the update is (tentatively) performed. Subsequently, each replica replies to the leader with an acknowledgement. The leader waits until enough replicas acknowledge the "update-table" message. When enough replicas acknowledge the "update-table" message the location server sends a positive acknowledgement to the client indicating that the update operation was performed. Finally, the leader initiates a second phase of cooperation with the replicas with the following results: 1) the updates to the replicated location table are permanently installed, and 2) all the pertinent locks held by the transaction executing the "update-table" command are released. Note that the second phase occurs after the leader replies to the client and thus it does not delay the execution of the transaction.

Now that we have discussed the first mechanism let us examine how this mechanism meets its objective. Recall that the motivation for building this mechanism was to enable execution of transaction operation at only one replica. This will result in a correct behaviour only if this replica has up-to-date information about the object upon which the operation is performed. Through the "server?" query the client can be informed of a proper replica that could correctly execute the transaction operation. Through the "update-table" command the client ensures that the location table reflects the most recent information. In this way, the next query of the location server will return the correct information. The discussion presented so far does not, however, preclude the possibility of a transaction using stale location information to access an object. The reason for this staleness lies in timing problems with regard to the execution of "server?" and "update-table" commands issued from different transactions. Remedies for this uncommon phenomenon will be discussed at the end of this section. It suffices to say that no incorrect transaction behaviour results due to the occurence of the above phenomenon.

Earlier we touched on the need for replicating the location server. Thus, a replication method is needed to guarantee correct replica coordination. In principle, any one of the well-known one-copy serializable techniques could be used to implement the replicated location service. However, our method has an additional performance constraint: transaction execution in a system employing our method should incur a delay similar to that of non-replicated systems. This, in turn, implies that the cost of obtaining the location information in a replicated system must be similar to that in a non-replicated distributed system. Therefore, we require that location service queries not require access to more than a single copy of the appropriate location table. Using a simple quorum technique would suggest that updates to the entries of the location table should be applied to all copies, since the read quorum must be one. This not only increases the performance cost of "update-table" operations but also limits their availability. Although it is evident that the performance trade-off between queries and updates is inevitable (recall that read and write quorums must intersect), it is not necessary to sacrifice availability. El Abbadi [Abbadi86, Abbadi89] presented a one-copy serializable replication method in which reading from only one copy is made possible without worsening the availability of the replicated service. The availability of read and write operations is defined by the read and write accessibility thresholds. Using these variables one can enforce the desired levels of availability independent of the quorum sizes. In addition, this method dynamically adjusts to network partitions so that operations remain available. For these reasons we consider this method to be appropriate for our location service.

3.3 Preemptive Optimistic Concurrency Control

Our second mechanism is developed so that our method can enjoy inexpensive transaction synchronization. Recall that the method in this paper requires that each operation be executed at only one site. The goal is to avoid replicated lock acquisition without introducing serializability and deadlock problems. Serializability problems may surface since transaction operations execute at only one site and since there is no guarantee that a conflicting operation from another transaction will execute at the same site. Thus, there exists the danger of undetected conflicting locks. Note that this danger does not exist in the viewstamped replication method since the conflicting transactions will be serialized at the primary replica. Deadlock problems may surface since conflicting transactions may obtain locks at different replicas and each may be waiting for the other to release them. Again, this is not possible in viewstamped replication since the transaction that has the lock at the primary site will always be allowed to proceed.

What follows is a description of the three major features of the concurrency control mechanism, employed by the location-based method, along with a short informal justification for its correctness (the method will be proved correct in a later section). There are three major characteristics:

- *optimism*
- *off-line replicated lock acquisition*
- *preemption*

Our concurrency control algorithm belongs in the general category of two-phase locking (2PL) algorithms, since the lock-acquisition phase is distinct from the lock-release phase. Concurrency control is optimistic in that it assumes that lock conflicts are rare. This assumption may be justified by considering the relatively short transaction execution times and the large number of objects within the system. The appropriate lock is only obtained at the replica where the transaction operation is executed. During 2PC, at the prepare phase, the appropriate locks will be obtained at a majority of the replicas for the object. We call this feature off-line replicated lock acquisition. Note that since majorities intersect, conflicting locks will be detected at prepare time.

Our concurrency control mechanism is preemptive. When a conflict between two transactions is detected the transaction with the lower priority is aborted. The abortion of transactions precludes the deadlock possibility, since transactions are not allowed to wait in order to acquire locks. In addition, serializability problems cannot occur since the off-line replicated lock acquisition feature guarantees that conflicting locks are always detected and the preemption feature of the mechanism guarantees that a transaction that would create serializability problems will not be allowed to commit.

Transaction priorities can be easily assigned in a manner that would enforce a total order among them. Since for every transaction there exists a unique client the transaction priority may be defined to be the transaction id appended to the site id hosting the client process. The transaction id is, in turn, composed of two fields: the client process id followed by the transaction count. The latter is simply a counter maintained by each client process which is incremented every time a new transaction is started at this client.

Having described the two major mechanisms incorporated in the optimistic, location-based replication method we can now proceed to describe the method.

3.4 The Replication Method

We describe our method by examining the behaviour of the main processes during transaction processing.

Processing at the client

Clients are responsible for finding the appropriate replica group for each transaction operation. This is accomplished by sending the "server?" query to the location server with arguments the objects that are referenced in the transaction. Subsequently, the client will select from the replicas returned by the location server one replica, (hereafter referred to as the *leader*) for each object, to which it will submit a request for executing the operation. Clients are also the coordinators of the two-phase commit protocol. For simplicity assume that clients are not replicated. After a client has received the replies back from the leaders, for all requests, then it starts the execution of the two-phase commit protocol. In the first phase, a client transmits prepare messages to each leader that communicated with it. Each leader eventually replies with an indication of whether it agrees to prepare and, if it is involved in updating an object, it also includes in the same message a list of the replicas that participated in the update. If all leaders agree to prepare then phase two (the commit phase) is initiated. The client sends an "update-table" message to the location server. This message contains a list identifying, for each object, which replicas store the recently-updated state. Subsequently, the client reliably broadcasts a commit message and waits until all leaders acknowledge it and the location server acknowledges the "update-table" message.

If some leaders refuse to prepare, the transaction is aborted by (reliably) sending an abort message to all the leaders that communicated with the client.

Processing at the leader

Upon reception of a request, the leader will start serving it, acquiring (locally) the required locks. As soon as the results are computed they will be immediately returned to the client. The propagation of the updated object states occurs in the background. We assume, at this point, the existence of an *update propagation process*. Its main task is to maintain a buffer of messages containing updates that should be communicated to the other replicas. This process operates periodically, attempting to transmit the updated object states to the replicas. It also maintains a *response list* containing the ids of those replicas that have acknowledged the reception of the updates. The leader interacts with the update propagation process through the message buffer. Before replying to the client, the leader enqueues the updated state to the communication buffer. When a sub-majority (i.e., a majority less one) of replicas have acknowledged the updates, the appropriate messages are removed from the buffer by the update propagation process.

After receiving a prepare message, the leader is responsible for verifying that a sub-majority of the replicas know of all the updates that it has performed on behalf of the preparing transaction. This is accomplished by having the leader examine the response list for a sub-majority of responses for each update. Actually, the leader may first rebroadcast the updates to the replicas and then check the response list. It is likely that, by this time, a sub-majority of replicas have already received and acknowledged the updates. Thus, typically, the waiting time for the leader will be small. When and if the leader receives a sub-majority of acknowledgements from the replicas it replies with a "prepared" message to the client. This message will also contain the list of replicas that updated each object. Otherwise, it refuses to prepare, sending a "refuse" message to the client.

After receiving a "commit" message from the client the leader will broadcast a "commit" message to the replicas, write the pair (transaction id, "committed") to the event history log, and wait until a sub-majority of replicas acknowledge the message. At this point, the leader will release any pertinent locks and reply with a "committed" message.

Processing at other replicas

Replicas that are not leaders are also involved in the processing of the transaction. This involvement takes place during the execution of the 2PC protocol and during update propagation. When a replica receives an update propagation message, it obtains the required locks for the objects that will be accessed. Subsequently, it will perform the indicated operations and then acknowledge the update messages (or the "prepare" message if it has been received). In the case of read operations the replica will only obtain the read lock — actual reading of the object is not required.

When a replica receives a "commit" message from the leader it appends a (transaction id, "committed") entry to its event history log. At this stage the replica will also release any locks that it holds pertaining to the committing transaction. Subsequently, it will reply to the leader with a "committed" message.

Discussion

As mentioned earlier, the description of the method given so far does not preclude the possibility of a transaction executing with stale location information. The following scenario can lead to this phenomenon. A client for a transaction may receive up-to-date location information, but before the client uses it to select a leader for an operation it is made obsolete due to an "update-table" operation executed by another transaction. Before we proceed to discuss the above scenario in more detail let us make two observations. First, all other timing problems regarding transaction commands to the location server can be resolved by the concurrency control algorithm employed by the replication method. In other words, if there is a conflict at a location table entry, the same conflict will be detected at the object represented by that entry and one of the two transactions will abort. This ensures that the transactions which are allowed to commit will always have obtained the most recent information from the location service. Second, the above scenario that results in a transaction executing with stale location information is very rare. The reason for this is partly due to the first observation. In addition, however, it is instructive to see that "update-table" commands occur as a result of some failure in the replica group for a replicated object. The reasonable assumption that failures are infrequent implies that "update-table" commands are infrequent. Furthermore, the list of up-to-date sites for an object x returned in the "server?" query will likely have many sites in common with the list of sites for x included in any "update-table" message. Thus, it is likely that a client will indeed select a leader that has the most recent state for the referenced object even if an "update-table" command has been executed before the selection of the leader.

It is important to ensure correct transaction behaviour in the presence of the above anomaly. Correctness can be easily guaranteed by using one of the following two mechanisms. Recall that our method requires transactions to hold read locks at location table entries until they commit. In this way, conflicts with "update-table" commands from another transaction will be detected. The first mechanism requires the transaction holding the read lock to be aborted, thereby giving precedence to commiting transactions. The disadvantage of this approach is that if read locks at the location table are kept until commit time, dummy "update-table" messages are required to release them.

The second mechanism is based on the earlier observations recognizing the rareness of the above anomaly. Therefore, it does not require the holding of read locks at location table entries until commit time. Instead, the client releases them immediately after the results to the "server?" query are received. The event that the location information received by the client has become obsolete due to a subsequent "update-table" operation of another transaction is detected and dealt with as follows: Each replica site is required to store a version number as part of the state of an object. When a leader propagates an update to a replica the update includes the new version number. If the leader's version number for the updated object is less than or equal to the version number of some replica for the same object then the transaction is aborted. Thus, transactions receiving stale location information can never commit.

Having introduced the method and its individual components along with some informal correctness arguments we now present a formal proof of the method.

4 Correctness Proof

The replication method of the previous section is now proved correct. The formal model and proof technique are taken from [Bernstein83]. We briefly introduce serializability theory for non-replicated systems and, subsequently, the extensions made by Bernstein and Goodman to obtain the theory for replicated systems.

4.1 Correctness in Non-replicated Systems

Transaction executions are modeled by *logs*. For every read (write) operation on object x of a transaction i a $r_i(x)$ ($w_i(x)$) entry appears in the log of i. Two operations conflict if they access the same object and at least one of them is a write. A transaction log is a poset $T_i=(\Sigma_i, <_i)$, with Σ_i representing the operations in transaction i and $<_i$ depicting the order in which these operations are performed. A database log over a set of transactions $T=\{T_1,T_2,\ldots,T_n\}$ is modeled by a poset L with the following properties: 1) $L=(\Sigma,<)$, with $\Sigma = \bigcup_{i=0}^{n} \Sigma_i$; 2) $< \supseteq \bigcup_{i=0}^{n} <_i$; 3) for every $r_i(x)$ there exists at least one $w_j(x)$ such that $w_j(x) < r_i(x)$; 4) all pairs of conflicting operations are related through $<$.

Transaction T_i *reads-x-from* transaction T_j if 1) L contains $r_i(x)$ and $w_j(x)$ entries, 2) $w_j(x)<r_i(x)$ in L and there exists no transaction T_k with $w_j(x)<w_k(x)<r_i(x)$. Two database logs are said to be equivalent if they have the same read-x-from relation for all objects x.

A log is serial if it is totally ordered and for any two transactions T_i and T_j either all operations of T_i precede all operations of T_j or vice versa. A database log is serializable if it is equivalent to a serial log. The serialization graph of a log L, $SG(L)$, is a graph such that $SG(L)=(V,E)$, with $V=\{T_1,\ldots,T_n\}$ and $E=\{T_i \rightarrow T_j \mid$ there exist conflicting operations op_i, op_j with $op_i<op_j\}$.

Theorem 1 [Papadimitriou79]: A concurrency control algorithm for non-replicated databases satisfies the serializability correctness criterion if *SG(L)* is acyclic.

4.2 Correctness in Replicated Systems

The serializability theory for non-replicated databases is easily extended to the replicated case. The extensions are needed in order to formalize the distinction between a logical replicated object x and the copies of it. In general, we say that a replicated object x, has the (physical) copies denoted by x_{a_1}, x_{a_2}, ..., x_{a_t}, where a_i, $i=1,\ldots,t$ represent the sites that store a copy of x. Two operations are now said to conflict if at least one of them is a write and they access the same physical copy. Thus, we have two kinds of operations, logical and physical, and a translation function which, given a logical operation as an argument, returns a set of physical operations needed to carry out that operation in a replicated system. Formally, the translation function t, is defined as: $t(r_i(x)) = r_i(x_a)$ for some site a storing a copy of x; and $t(w_i(x))=\{w_i(x_1)$, $w_i(x_2)$, ..., $w_i(x_m)\}$, where m copies are enough to guarantee correctness.

A replicated database log (*rd log*) over a set of transactions is a poset L with the following properties: 1) $L=(\Sigma, <)$, with $\Sigma=t(\bigcup_{i=0}^{n} \Sigma_i)$; 2) For each T_i and op_{i1}, op_{i2} if $op_{i1}<op_{i2}$ then each physical operation in $t(op_{i1})$ is $<$-related to $t(op_{i2})$; 3) for every $r_i(x_a)$ there exists at least one $w_j(x_a)$ with $w_j(x_a)<r_i(x_a)$; 4) all pairs of conflicting operations are related through $<$.

A transaction T_i *reads-x-from* transaction T_j in a replicated system if an rd log L for this system contains $r_i(x_a)$, $w_j(x_a)$ for some copy x_a, with $w_j(x_a)<r_i(x_a)$ and there exists no T_k with $w_j(x_a)<w_k(x_a)<r_i(x_a)$.

An rd log is one-copy serializable if it is equivalent to a serial *one-copy* log. The other definitions remain as before. The one-copy serialization graph for an rd log L, $1\text{–}SG(L)$, is constructed as follows: 1) it contains the serialization graph of L, $SG(L)$; 2) it embodies a total order of all transactions that write object x, denoted by $\ll_x$, for all objects x, and 3) for each x and transactions T_i, T_j and T_k such that T_j reads-x-from T_i and $w_i(x) \ll_x w_k(x)$, $1\text{–}SG(L)$ contains a path from T_j to T_k. This path is called a *reads-before* path and models the need for enforcing $r_j(x)$ to precede $w_k(x)$.

Theorem 2 [Bernstein83]: For an rd log L, if $1-SG(L)$ is acyclic then L is one-copy serializable.

This is the main tool for proving that replication methods are correct.

4.3 The Proof

Since we use the virtual partitions method for the replicated location server we are guaranteed that the location table is one-copy serializable. In addition, via the use of version numbers or the holding of read locks at location table entries until commit time the method can detect all events in which a transaction would execute with stale location information. Since our method aborts such transactions it is guaranteed that no transaction would execute with stale location information. Thus, we know that the "server?" query will always return the correct information. Knowing this, we concentrate on proving the one-copy serializability of the replicated objects handled by the location-based method without modeling the interaction of the client with the location service.

The proof requires four basic steps: 1) formalize the behaviour of the replication method using an rd log L; 2) construct the serialization graph $SG(L)$; 3) construct the one-copy serialization graph $1-SG(L)$, and 4) show that $1-SG(L)$ is acyclic.

Using the definition of an rd log that was presented previously and the description of the replication method we construct an rd log L, corresponding to transaction executions in a system using our method. L consists of

1) For every transaction T_i that reads object x at replica site a, L contains $rd-lock_i(x_a) \rightarrow r_i(x_a) \rightarrow \{rd-lock_i(x_1),\ldots,rd-lock_i(x_m)\}$, where $\lfloor \frac{n}{2} \rfloor \leq m \leq n-1$, with n representing the total number of copies of x. This defines $t(r_i(x))$. The rd—lock operation represents the acquiring of read locks on the indicated object.

2) For every transaction T_k that writes an object x, L contains $wr-lock_k(x_a) \rightarrow w_k(x_a) \rightarrow \{(wr-lock_k(x_1) \rightarrow w_k(x_1)),\ \ldots,\ (wr-lock_k(x_m) \rightarrow w_k(x_m))\}$, with m defined as above. This defines $t(w_k(x)$. The wr—lock operation represents the acquiring of write locks.

3) If $r_i(x_a) \in L$ and T_i subsequently writes x then $r_i(x_a) \rightarrow \{w_i(x_1),\ldots,w_i(x_m)\}$.

4) Every $r_i(x_a)$ follows at least one $w_j(x_a)$, where $i \neq j$.

5) All pairs of conflicting transactions are $<$-related: i) $rd-lock_i(x_a)$ conflicts with $wr-lock_j(x_a)$, and ii) $wr-lock_i(x_a)$ conflicts with $wr-lock_j(x_a)$ and $rd-lock_j(x_a)$, for some copy x_a and transactions T_i, T_j.

Rules 1) and 2) define the translation function t with respect to the specification of our replication method. Having introduced all the necessary physical operations we simply follow the definition of an rd log to construct one that corresponds to allowable executions of transactions in our method.

The construction rules for $SG(L)$ have already been explained and will not be repeated here. The following two lemmas play a central role in the proof.

Lemma 1: *If* $T_i \rightarrow T_j \in SG(L)$ then T_i committed before T_j.

Proof: Since $T_i \rightarrow T_j \in SG(L)$ we know that there exists at least one pair of operations op_i, $op_j \in L$, such that $op_i < op_j$ and op_i conflicts with op_j. The conflict implies the existence of a common copy on which both operations were performed. This, in turn, implies that if the two transactions executed concurrently (e.g., op_j occurred before T_i committed) the conflict would be detected. Recall that the concurrency control algorithm employed in our method forces the transaction with the lower priority to abort. If we assume that T_i and T_j executed concurrently then since op_i is ordered before op_j, T_j would be the aborted transaction and it would abort before op_j was performed. But this would imply that L does not contain the entry op_j (recall that L contains only the allowable transaction operations). Therefore, we can conclude that T_j acquired the locks(s) necessary for performing op_j after T_i had released them, that is, after T_i committed. Thus T_i committed before T_j. □

Lemma 2: The serialization graph *SG(L)* is exactly the same as the one-copy serialization graph, $1\text{–}SG(L)$.

Proof: To construct $1\text{–}SG(L)$ we first construct $SG(L)$ and we subsequently add enough edges so that i) there exists in $1\text{–}SG(L)$ a total write order (denoted $\ll_x$) among all transactions writing object x, for all objects x, and ii) for all T_i, T_j, T_k such that T_j reads-x-from T_i and $T_i \ll_x T_k$ we must have a (reads-before) path from T_j to T_k.

First we claim that $SG(L)$ already embodies a total write order for all objects x. This is easily shown since majorities intersect. Our method requires that logical writes be physically applied to a majority of the copies of the updated object. This implies that for any two writes on any object x, $w_i(x)$, $w_j(x)$, there must exist a copy x_a such that $w_i(x_a) \in L$ and $w_j(x_a) \in L$. The existence of this copy implies, by definition, that the two writes conflict (via the wr—lock operations). From the construction of L we know that any pair of conflicting operations are $<$-related. Finally, from the construction rules of $SG(L)$ we know that there exists an edge between the nodes T_i and T_j.

Therefore, for any object x and for any two transactions writing x, either $T_i \rightarrow T_j \in SG(L)$, or $T_j \rightarrow T_i \in SG(L)$. Thus, $SG(L)$ contains a total write order for all objects, as required.

Second, we claim that no edges need be added to the edges of $SG(L)$ to create the reads-before paths. We actually make a stronger claim, namely, if T_j reads-x-from T_i and $T_i \ll_x T_k$ then $T_j \rightarrow T_k \in SG(L)$, for any such transactions T_i, T_j, T_k and any object x.

To show that the second claim holds recall that the method requires that read locks be obtained at a majority of replicas (during the prepare phase) for every read operation. Write locks are also obtained at a majority of replicas. Recall that rd-lock operations conflict with wr-lock operations, provided they are applied on (at least) one common copy. Since majorities intersect and T_j reads x and T_k writes x we must have, for some copy x_a, that either $r_j(x_a)<w_k(x_a)$ in L or vice versa. If we assume that $w_k(x_a)<r_j(x_a)$ then $T_k \rightarrow T_j \in SG(L)$. From Lemma 1 we then conclude that T_k commits before T_j. However, it is given that $T_i \rightarrow T_k \in SG(L)$, since $T_i \ll_x T_k$. The last two statements then violate the assumption that T_j reads-x-from T_i, because if T_i committed before T_k and T_k committed before T_j then T_j would read x from T_k. Thus, we have reached a contradiction. Therefore our assumption was wrong. Hence, $r_j(x_a)<w_k(x_a)$ which implies that $T_j \rightarrow T_k \in SG(L)$.

From the above two claims it is easily seen that Lemma 2 holds. □

Having proved these two lemmas the main theorem follows, easily.

Theorem 3: $1-SG(L)$ is acyclic.

Proof: (by contradiction)

Lemma 2 implies that the statement of Lemma 1 holds also for $1-SG(L)$. Assume that $1-SG(L)$ contains a cycle, say, $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_r \rightarrow T_1$. From Lemma 1 we know that T_1 commits before T_2, ..., commits before T_r. Since the "commits-before" relation is obviously transitive and irreflexive we obtain the contradiction that T_1 commits before T_r and T_r commits before T_1. Thus, our assumption that a cycle exists was wrong. □

5 Comparison with Other Work

In this section we concentrate on the performance and availability characteristics of the method described in this paper. We also compare this method to other well-known methods.

5.1 Performance Considerations

Our method was developed out of our desire to minimize the cost of replication in distributed systems. This, in turn, translates to minimizing transaction execution delay. Naturally, the goal was to achieve transaction delays that are similar to those occurring in one-copy systems. This is only the second replication method that can make this claim—the other being the viewstamped replication method.

Let us discuss the relative performance merits of the location-based method and the viewstamped method and at the same time indicate why they have similar performance to non-replicated systems. Before a transaction starts executing, the client requests location service. In a non-replicated system this typically involves communicating with a remote site that stores the location information. In viewstamped replication the client again obtains from a remote site information regarding the replica group that implements the referenced objects. Subsequently, the client consults enough of these replicas in order to determine the primary replica. Determining the primary may be costly and for this reason information regarding primaries of replica groups is cached. However, the success of cache memories depends largely on the property of locality of reference. Locality of reference has two dimensions: *temporal locality* and *spatial locality*. Temporal locality refers to the property that recently-cached information will be needed again by the client in the near future. Spatial locality assumes that information is cached in blocks and that a lot of the information contained in a recently-cached block will soon be needed. In the viewstamped method, information is not cached in blocks and thus the spatial locality property cannot hold. In addition, we do not believe that temporal locality holds. It is, in general, doubtful that transactions that access the same replicated objects will execute at a client within a short time period of one another. Therefore, we believe that locating the primary replica may be a significant source of performance degradation. This degradation will be more severe for transactions involving a large number of objects.

After obtaining the necessary location information, the client submits each operation to an appropriate site that can execute it. In non-replicated systems, as in systems that employ the viewstamped and location-based methods, the operation is executed at only one site and the results are immediately sent back to the client. In all three cases any concurrency that might be possible within a transaction may be easily exploited. In the background, however, location-based and viewstamped systems broadcast the effects of the executed operation to the other replicas. Both methods require only a majority of acknowledgements.

When all transaction operations have been performed the 2PC protocol is initiated. A site participating in 2PC in a non-replicated system need not communicate with any other sites before replying to the "prepare" and "commit" messages. It must, however, write transaction status information and the effects of transaction operations to stable storage. In both location-based and viewstamped methods an extra round of messages is needed instead of writing to stable storage. Recall that the extra round of message exchange is needed so that enough replicas are informed of the effects of the transaction operations. It is generally agreed [Birman85, Oki88] that stable-storage writes are approximately as costly as this additional messaging round. Here lies the essence of the claim that stable storage is not needed in a replicated system, but can be successfully substituted by the replicated information. Hence, we can see that the execution of the 2PC protocol will introduce similar delay in both viewstamped and location-based replicated systems as in non-replicated systems.

We should point out that the location-based method adds one more participant to the 2PC process. In essence, the delay during the execution of the 2PC protocol is equivalent to the delay that would occur in the viewstamped method if one more primary had to be included in the 2PC process. Thus, 2PC is slightly more expensive in our method than it is in the viewstamped method. However, we expect that failures will be rare. This implies that in most cases significantly more than a majority of replicas will know of an updated object state. Therefore, it is likely that the set of replicas that was returned to the client from the location server will be identical to the lists returned by the leaders to the client in the prepare phase. Thus, no "update-table" message need be sent. As a result, there will often not need to be an extra participant in the 2PC process.

As mentioned earlier, the location-based method avoids the disadvantages associated with replication methods that are based on the primary-site paradigm for replication. In particular, there exists the danger of *performance bottlenecks* being formed at the primary sites. Furthermore, our method can easily support *load balancing* since the client has the luxury of choosing a leader from a set of replicas.

No other replication method can claim performance similar to non-replicated systems. Typically, in other methods reads and/or writes are applied to a number of replicas before the results are obtained. A notable exception is the method employed by ISIS [Birman85]. ISIS uses the notion of a replica coordinator which is similar to our notion of a leader replica. Operations are only performed at the coordinator. However, an expensive two-phase protocol is used for replicated lock acquisition and avoiding deadlock problems.

5.2 Availability

The availability of operations in location-based replication is constrained by the requirement that a majority of replicas be available for each transaction operation to prepare and commit during 2PC. In this respect, the availability characteristics of our method are similar to those of viewstamped replication. We should note that requiring a majority of sites to be available is not very restrictive, since, in most cases, a majority of replicas will indeed be available.

Our method, can be easily modified so that, like most quorum-based methods, it is flexible enough to allow trade-offs between the availability of read and write operations. Read and write availability variables A_r and A_w are used instead of majorities of replicas. During the execution of transaction operations, as before, only one replica is accessed. However, during the prepare and commit phases of 2PC instead of contacting a majority of replicas a read operation accesses A_r replicas. Similarly, a write operation accesses A_w replicas. As long as $A_r+A_w>n$ and $2A_w>n$, where n is the total number of copies, correctness is guaranteed. In this way, by manipulating the variables A_r and A_w we can trade-off the availability characteristics of read and write operations. This is not possible in the viewstamped replication method since the view-change algorithm requires that a majority of replicas be involved in 2PC in order to ensure that the correct primary is selected. Therefore, location-based replication is more flexible than viewstamped replication.

A few recent research efforts have been directed towards developing methods whose availability characteristics degrade gracefully in response to failures and, especially, network partitions. For example, [Abbadi86, Abbadi89] present methods in which quorum sizes are adjusted (as failures occur) so that the availability of operations is unaffected. El Abbadi and

Toueg developed the concept of a view of a site which is an approximation of the set of sites with which communication is possible (virtual partition). Within each virtual partition read and write quorums may be redefined and operations will still be available provided there are more than a (globally defined) threshold value of accessible replicas. The method in [Abbadi86] disallowed concurrent writes in more than one partition. The method in [Abbadi89] fixes this deficiency. In [Herlihy87a], Herlihy uses a predefined set of quorum assignments. It consists of levels of different read and write quorum sizes. As failures and partitions occur and operations are made unavailable, given the current assignments, transactions can switch to another level in which the currently unavailable operation has a smaller quorum size and thus is available. This is called quorum inflation. All of these methods, however, require special algorithms (either view-change algorithms or quorum inflation/deflation algorithms). These algorithms are costly but on the assumption that partitions are rare or that availability is the primary concern, replicated systems may reasonably use such methods.

Availability is also affected by the locking mechanism which, in effect, makes objects temporarily unavailable. This is the case for most other methods. Location-based replication, as a result of the concurrency-control algorithm that it uses, induces transaction abortions when a lock conflict is detected. The drawback of this side effect is that some computation may occasionally be wasted. Recall that our design is based on the assumption that lock conflicts, in general, are rare. In comparison, viewstamped replication does not abort transactions when lock conflicts are detected but induces transaction abortions when nodes and communication links fail and a new view is initiated. Moreover, while the view-change algorithm is in progress, replicas are inactive and requests cannot be served. This disruption of service is even more pronounced since, after the new view and primary have been established, the client's cache must be updated (to reflect the new primary) before a transaction can start executing.

An important contribution made in [Abbadi86, Abbadi89] is the separation of availability concerns from performance concerns. Quorum sizes represent the performance variables used to define the performance cost while accessibility thresholds are the availability variables used to define the availability of operations. By fine tuning these variables the designer can address these issues independently. In our method, the performance variables have been optimized corresponding, in essence, to quorum sizes of one, while the availability variables can be set to reflect the desired levels of availability of read operations versus the availability of write operations.

6 Contributions and Concluding Remarks

The method presented here is a descendant of viewstamped replication. It is a modification and extension of that method, which in turn, is a modification and extension of the virtual partitions method. Viewstamped replication adopted the concept of a view, introduced a primary replica and incorporated the method into the transaction-processing mechanism. The introduction of a primary replica facilitates the minimization of the replication cost, while a view-change algorithm ensures correctness in the presense of failures. Location-based replication introduces two mechanisms that can be used in order to provide a similar performance to non-replicated systems without using the primary-site paradigm for replication. The first mechanism is a location service with an extended functionality. The key idea is the incorporation of this extended location service into the transaction-processing mechanism. The end result is that transaction operations need only execute at a single replica site. The second mechanism is a preemptive, optimistic concurrency control algorithm, that introduces an efficient way of avoiding serializability and deadlock problems.

Our method avoids the inefficiencies associated with methods based on the primary-site paradigm for replication. In particular, viewstamped replication may cause performance bottlenecks, disallows load balancing and may exhibit considerable disruption of service (especially in environments where failures are more frequent). In addition, location-based replication is simpler because it does not require any supporting algorithm, such as a view-change algorithm. It is also easily proved correct.

Our method has good availability characteristics, in that only a majority of replicas is required to be available. Location-based replication is also more flexible than viewstamped replication in that it can allow the availability characteristics of read and write operations to be traded-off. However, it is not as flexible as some of the aforementioned methods that make availability degrade gracefully with failures. We believe that there is no inherent restriction in our method that would preclude the above desirable property for availability. This is an important area for future research.

Bibliography

[Abbadi86] A. El Abbadi and S. Toueg, "Availability in partitioned replicated databases (extended abstract)", Proc. of the 5th Symp. on Principles of Database Systems, March 1986, pp 240-251.

[Abbadi89] A. El Abbadi and S. Toueg, "Availability in partitioned replicated databases", *ACM Transactions on Database Systems*, June 1989, vol. 14, no. 2, pp 264-290.

[Bernstein81] P. Bernstein and N. Goodman, "Concurrency control in distributed database systems", *ACM Computing Surveys*, vol. 13, no. 2, June 1981, pp 185-222.

[Bernstein83] P. Bernstein and N. Goodman, "The failure and recovery process for replicated databases", Proc. 2nd ACM Symp. on Principles of Distributed Computing, August 1983, pp 114-122.

[Bernstein84] P. Bernstein and N. Goodman, "An algorithm for concurrency control and recovery in replicated distributed databases", *ACM Transactions on Database Systems*, vol. 9, no. 4, December 1984, pp 596-615.

[Birman85] K. Birman, "Replication and fault-tolerance in the ISIS system", Proc. of the 10th ACM Symposium on Operating System Principles, December 1985, pp 79-86.

[Demers 87] A. Demers, et al, "Epidemic algorithms for replicated database maintenance", Proc. of 6th Symposium on Principles of Distributing Computing, August 1987, pp 1-12.

[Eager83] D. Eager and K. Sevcik, "Achieving robustness in distributed database systems", *ACM Transactions on Database Systems*, vol. 8, no. 3, September 1983, pp 354-381.

[Gifford79] D. Gifford, "Weighted voting for replicated data", Proc. of the 7th Symposium on Operating System Principles, December 1979, pp 150-162.

[Hammer80] M. Hammer and D. W. Shipman, "Reliability mechanisms in SDD-1: A system for distributed databases", *ACM Transactions on Database Systems*, vol. 5, no. 4, December 1980, pp 431-466.

[Herlihy85] M. Herlihy, "Comparing how atomicity mechanisms support replication", Proc. of the 4th ACM Symposium on Principles of Distributed Computing, August 1985, pp 102-110.

[Herlihy86] M. Herlihy, "A quorum consensus method for abstract data types", *ACM Transactions on Computer Systems*, vol. 4, no. 1, February 1986, pp 32-53.

[Herlihy87a] M. Herlihy, "Dynamic quorum adjustment for partitioned data", *ACM Transactions on Database Systems*, vol. 12, no. 2, June 1987, pp 170-194.

[Herlihy87b] M. Herlihy, "Concurrency versus availability: Atomicity mechanisms for replicated data", *ACM Transactions on Computer Systems*, vol. 5, no. 3, August 1987, pp 249-274.

[Kohler81] W. H. Kohler, "A survey of techniques for synchronization and recovery in decentralized computing systems", *ACM Computing Surveys*, vol. 13, no. 2, June 1981, pp 149-185.

[Lampson81] B. Lampson, "Atomic transactions", Lecture notes in Computer Science Vol. 105: *Distributed Systems: Architecture and Implementation*, Springer Verlag, Berlin 1981, pp 246-265.

[Minoura82] T. Minoura and G. Wiederhold, "Resilient extended true-copy token scheme for a distributed database system", *IEEE Transactions on Software Engineering*, vol. 8, no. 3, May 1982, pp 173-188.

[Oki88a] B. M. Oki and B. Liskov, "Viewstamped replication: A new primary copy method to support highly available distributed systems", Proc. of the 7th ACM Symposium on Principles of Distributed Computing, August 1988, pp 8-17.

[Oki88b] B. M. Oki, "Viewstamped replication for highly available distributed systems", Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, August, 1988. Also published as MIT/LCS/TR-423.

[Oppen83] D. Oppen, and Y. K. Dalal, "The clearinghouse: A decentralized agent for locating named objects in a distributed environment", *ACM Transactions on Office Information Systems*, vol. 1, no. 3, July 1983, pp 230-253.

[Papadimitriou79] C. Papadimitriou, "The serializability of concurrent updates", *Journal of the ACM*, vol. 26, no. 4, October 1979, pp 631-653.

[Reed83] D. P. Reed, "Implementing atomic actions on decentralized data", *ACM Transactions on Computer Systems*, vol. 1, no. 1, February 1983, pp 3-23.

[Shroeder84] M. D. Shroeder, A. D. Birell, and R. M. Needham, "Experience with grapevine: The growth of a distributed system", *ACM Transactions on Computer Systems*, vol. 2, no. 1, February 1984, pp 3-23.

[Taylor86] D. J. Taylor, "How big can an atomic action be", Proc. of 5th Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp 121-124.

[Thomas79] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases", *ACM Transactions on Database Systems*, vol. 4, no. 2, June 1979, pp 180-209.