The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed Systems

Peter Triantafillou, *Member, IEEE*, and David J. Taylor, *Member, IEEE*

***Abstract*— Replication techniques for transaction-based distributed systems generally achieve increased availability but with a significant performance penalty. We present a new replication paradigm, the *location-based paradigm*, which addresses availability and other performance issues. It provides availability similar to quorum-based replication protocols but with transaction-execution delays similar to one-copy systems. The paradigm further exploits replication to improve performance in two instances. First, it takes advantage of local or nearby replicas to further improve the response time of transactions, achieving smaller execution delays than one-copy systems. Second, it takes advantage of replication to facilitate the independent crash recovery of replica sites—a goal which is unattainable in one-copy systems. In addition to the above the location-based paradigm avoids bottlenecks, facilitates load balancing, and minimizes the disruption of service when failures and recoveries occur. In this paper we present the paradigm, a formal proof of correctness, and a detailed simulation study comparing our paradigm to one-copy systems and to other approaches to replication control.**

***Index Terms*— Availability, concurrency control, distributed computing, partitionings, recovery, replication, transactions.**

I. Introduction

IN THIS PAPER we use the concept of *transactions* [20] to describe the execution of programs. Transactions consist of primitive operations (read, write) and are characterized by the properties of *serializability* [24] and *atomicity* [2]. Different researchers have assigned different properties to transactions (see [26] for a related discussion) but they amount to serializability and atomicity. Informally, serializability ensures that any concurrent execution of transactions that is allowed by a correct concurrency control algorithm is equivalent to some serial execution of them. Atomicity ensures that transactions have an all-or-nothing effect. Within the context of replicated systems, transactions must obey the *one-copy serializability* [3] correctness criterion. Informally, the additional requirement introduced by the latter criterion is that replicated objects behave as if they were not replicated.

Distributed systems consist of computing nodes that are interconnected by a communication network. Site crashes and communication-link failures may make data inaccessible. In addition, accessing remotely-stored data can be very costly. For these reasons, replication of data objects is deemed desirable. By storing redundantly the state of objects a system can provide access to a replicated object even though a number of the replica sites are inaccessible. Furthermore, by placing copies of an object physically close to the sites where operation requests for the object originate, the cost of remote data accesses can be significantly reduced. On the other hand, in order to guarantee correctness it is required that replicas collaborate. Such collaboration typically involves multiple rounds of message exchange and thus incurs a significant cost. The latter cost is one of the main reasons for the lack of replicated databases that satisfy the one-copy-serializability correctness criterion.

The balance of the paper is organized as follows. Section II presents the motivations and goals of this work. Section III presents the fundamental components of the replication paradigm. Section IV presents a detailed description of the replication-control protocol. In Section V a proof of correctness is given. Section VI discusses the effect of failures in the performance of the location-based paradigm and discusses the crash-recovery algorithm of the paradigm. Section VII details a simulation study comparing the performance of the paradigm with other well-known approaches to replication control. Section VIII presents a qualitative comparison with related work. Section IX outlines the contributions made by this work and contains concluding remarks.

II. Motivations and Goals

This research addresses the problem of replication control in transaction-processing systems, which enforce one-copy serializability and the atomicity of transactions, and which make no assumptions about the applications. Thus, systems and protocols which enforce other notions of consistency (such as [5]) or which place semantic constraints on the applications (such as [19]) will not be considered.

The introduction of replication into distributed systems is necessary if they are to provide availability and fault tolerance. However, although considerable research effort has been directed towards the design of replication-control protocols, replication is still viewed as a 'necessary evil'. Related work has concentrated on increasing availability, which is typically achieved by introducing special mechanisms that have a multi-

Manuscript received September. 1992; revised June 1994. This work was supported by the Natural Science and Engineering Research Council of Canada under a postgraduate scholarship and Grant OGP0003078.

P. Triantafillou is with the School of Computing Science, Simon Fraser University, Burnaby, B.C., V54 1S6, Canada.

D. J. Taylor is with the Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada.

IEEE Log Number 9407722.

faceted adverse impact on the performance of the system. One typical example of such a mechanism is that client processes executing operations on replicated data must access synchronously many replicas for each operation. Another typical example is the need for special system transactions that execute whenever failures or recoveries take place in order for the replicas to maintain consistent information. We believe that the resulting increase in transaction-execution delays and the wasting of system resources, such as network bandwidth and CPU cycles, imposed by these protocols is one of the principal reasons for the lack of commercially-available replicated transaction-based distributed systems. As a result, our research efforts were principally motivated by the perceived need to address performance, as well as availability, issues. We wish to improve the availability of distributed-system services without paying significant performance penalties. In fact, we wish to employ replication not only as a means to improve the availability of distributed-system services but also to improve the efficiency of rendering the services.

Our thesis is that if a replication-control protocol exhibits transaction delays similar to those in nonreplicated systems then, given the availability benefits, a significant step will have been taken towards making replicated distributed systems a reality. It is, however, inappropriate to concentrate exclusively on transaction-execution delays. Other aspects of performance must be taken into account. One such aspect is the availability afforded by a protocol. Another is the ability of a protocol to facilitate load balancing and avoid bottlenecks in distributed systems. In addition, one may consider the smoothness of service provided by the protocol. In other words, how does the system behave under expected anomalous events such as failures and recoveries? This is perhaps more significant than it may appear at first sight. Although anomalous events are rare, chaos should not be caused when they occur and the replication service should continue to be provided without significant performance penalties—especially to clients that are not directly affected by these events.

We therefore set out to design a replication-control protocol with the following goals.

1) Achieve transaction-execution delay similar to one-copy systems.
2) Ensure graceful degradation of availability and other performance metrics in the presence of failures.
3) Not introduce performance overhead when beneficial events, such as the recovery of a site or the repair of a communication link, occur.

III. Fundamental Components

It is obvious that, if our goals are to be achieved, a transaction client should be forced to wait for only a single replica to acquire a lock and execute the transaction operation. We must then deal with two issues. First, the method must ensure that serialization problems do not emerge from the above requirement for nonreplicated synchronous lock acquisition. Second, since a client will obtain the results of operations that are executed at a single replica site, a client should submit the operation to a replica that can correctly execute it (i.e., a replica that stores an up-to-date copy of the object). Therefore, we focused our attention on developing.

1) a mechanism that allows a client to access only a single replica (called a leader) and still obtain the most recent state for the referenced data object, and
2) an inexpensive transaction-synchronization mechanism that avoids the need for synchronous, replicated lock acquisition.

Before we proceed with the description of the components of our paradigm, we first present the system model, define some terminology, and discuss transaction processing in the model.

A. The System Model

A replicated distributed system consists of many sites interconnected by a communication network. Data stored in it are replicated at a number of sites. As a result, replicated data objects are managed by a group of processes, called replicas, executing at the 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.

Our environment can witness processor, communication-link, and performance failures. Processors are *fail-stop*, i.e., when they fail they stop processing completely. Communication-link failures may induce *network partitioning* in which groups of sites are formed with individual groups being isolated from one another. Performance failures present the illusion that communication links have failed. They are typically manifested as time-out expirations which are caused by heavily loaded links. We assume that processor and link failures are not malicious.

We model computations in the distributed system with the concept of 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. At the site at which a transaction originates a process is responsible for executing the transaction. This process is referred to as the client and during the execution of a transaction the client may communicate with several 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, which typically is itself replicated for availability reasons. The location servers maintain the *location tables* which contain the desired information.

When all transaction operations have been executed and the client has received the results, it will initiate the two-phase commit (2PC) protocol [17]. 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 so that *strictness* [2] can be enforced.

The replicas within a replicated server (eventually) record the effects of transaction operations in a local *event history log* which is used for recovery.

The following information defines a replicated object state for an object x.

Replicated_Object_Record (x)
begin record
value: type_of(x);
version: integer,
A_r: integer,
A_w: integer,
replicas: set_of_sites;
end record

The value of an object depends on the type of the object. The integers $A_r(x)$ and $A_w(x)$ reflect the availability and conflict-detection variables that have to be satisfied in order for transaction operations on x to be completed. (These variables as we shall see are similar to Gifford's read and write quorums [9]).

We assume the existence of two types of broadcasting algorithms, synchronous (*bcast*()) and asynchronous (*Async_bcast*()). They differ in that *bcast*() forces the sender to block until all the members of the destination group have acknowledged the message, whereas *Async_bcast*() is a nonblocking primitive. The first parameter in these system calls represents the destination group of processes. Finally, the system call *write*() is used to append recovery information to the event history log (EHL).

op_lock (x) represents the acquisition of the appropriate lock (read or write) on object x, while *op* (x) represents the execution of the operation. *release_lock* (x) represents the releasing of a lock on the referenced object.

B. An Extended Location Service

Our replication paradigm is based on an extended location service and on the incorporation of this service into the transaction-processing facility.

In a nonreplicated distributed system, a client that is about to execute a transaction typically 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 the object. In a replicated distributed system, the location server may provide the client with a list of sites which are capable of performing an indicated operation.

In location-based replication, the client interacts with the location server to obtain a list of sites that contain an *up-to-date copy* of each replicated object referenced in the transaction.

As operations execute and failures and recoveries occur, there will be a changing set of up-to-date replicas. Thus, we must allow for updating the location tables so that the information provided by the location server to the client is correct. Our method guarantees that at commit time the client knows, for each updated object, which replicas were involved in the execution of the update. Thus, the client knows which replicas are currently up-to-date. If a new set of up-to-date replicas results, along with the commit messages, the client submits an update request to the location server. More precisely, the interface to 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));

The "server?" function is simply the query submitted by the client in order to find out the set of up-to-date replicas. "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.

Since the location server is vital to transaction execution, our design must avoid performance bottlenecks at the location server. In other words, "server?" and "update-table" operations should be performed as efficiently as possible (i.e., must hold locks for very short times).

Our extended location service is characterized by short lock-holding times and, in addition, the availability of data objects and their operations is not unnecessarily dependent upon by the availability of location-server replicas. Transactions executing in a replicated system that employs our method require only one location-server replica (so that location information can be obtained) and enough data-object replicas (so that one-copy serializability can be enforced) to be available.

1) Performing Location Services: The location table is primarily stored on nonvolatile storage (disk). Updates to it are only recorded on volatile storage which is periodically flushed to disk. In this way we avoid having the cost of disk writes create throughput problems at location-server replicas.

Location-table entries are locked using two types of locks, *s-locks*, which are associated with "server?" operations, and *u-locks* which are associated with "update-table" operations. s-locks conflict with u -locks and u-locks conflict with s-locks and u-locks. A location-server replica will execute a "server?"query as follows. First it attempts to obtain an s-lock (locally) on the appropriate location-table entry. If another transaction holds a u-lock on the same entry then the requesting transaction waits in a lock queue. If no other transaction holds a conflicting lock then the table entry is s-locked, read, and the result returned to the client. Subsequently, the s-lock is immediately released. Note that the "server?" operation is executed at only one location-server replica. This is in agreement with our goal of achieving performance similar to distributed systems that do not employ replication. With regard to availability, only one location-server replica is required to be available.

The "update-table" operation is submitted by the client during the commit phase of the 2PC protocol. Since the location server is replicated, it is desirable that the "update-table" operation execute at all location-server replicas and thus it is broadcast to all location-server replicas. A location-server replica performs the "update-table" operation as follows. For every object included in the "update-table" operation, a u-lock is locally placed on the relevant table entry, the update is performed, and the lock is immediately released. u-lock conflicts force transactions to wait in lock queues.

The broadcasting of operation and lock-acquisition requests to location-server replicas is achieved using a *nonatomic multicast protocol*. Atomic broadcasts typically require two

```
receive (sender, op(x)) ;
if op(x) = server? (x) then begin
   s_lock (Loc_Table[x]) ;
   send (sender, Loc_Table[x]) ;
   release_lock (Loc_Table[x]) ;
else
   /* incoming operation is an "update-table (x, set_of_replicas) " */
   u_lock (Loc_Table[x]) ;
   Loc_Table[x].sites = set_of_replicas ;
   release_lock (Loc_Table[x]) ;
endif ;
```

Fig. 1. A location-server replica in the location-based paradigm.

phases with each phase consisting of two rounds of messages. If locks are obtained during the first phase and released in the second phase, we risk long lock-holding times and thus the possibility of bottlenecks at the location server. This could degrade transaction response times. In addition, we wish the availability of data objects to be independent of the availability of location-server replicas. Atomic broadcasts conflict with this goal. If, for example, the leader location-server replica failed after the first phase but before initiating the second phase, no "server?" queries could be served because of the u-locks at the location-server replicas. Also note that the client is not waiting for replies to the "update-table" message.

However, it is possible that failures and/or concurrent update-table operations cause different location-server replicas to store different values for the same location-table entries. Thus, care must be taken to ensure that serialization problems do not result from having transactions obtain stale location information. Our method detects such nonserializable transactions and forces them to abort and restart. It is important to keep the number of such restarts small. In a later section we discuss why such restarts will be very rare.

Fig. 1 depicts the behavior of location-server replicas.

C. Asynchronous Concurrency Control

The second component of our paradigm allows inexpensive transaction synchronization. The goal is to avoid *synchronous* replicated lock acquisition without introducing serializability problems. Serializability problems may surface since each transaction acquires locks (synchronously) at only one site. Thus, there exists the potential of undetected conflicting locks.

The major characteristic of our concurrency control mechanism is *asynchronous replicated lock acquisition*. The concurrency-control algorithm belongs in the general category of two-phase locking (2PL) algorithms (i.e., the lock-acquisition phase is distinct from and precedes the lock-release phase). The novelty of the algorithm is that the appropriate lock is synchronously obtained only at one replica, called the leader replica. After the operation is executed, the leader replica sends an acknowledgment to the client and (asynchronously) broadcasts lock-acquisition messages to the other replicas. Upon receiving such a message a nonleader replica will (locally) obtain the lock. Thus, nonleader replicas acquire locks requested by a client in parallel with the client's execution of subsequent operations.

Since in a system with replicated data locks are typically acquired at more than one replica, deadlocks may occur. Our concurrency control mechanism can employ any strategy for dealing with deadlocks, such as aborting a conflicting transaction, or allowing a transaction to wait for a lock and employ a deadlock detection and resolution algorithm. For simplicity, we initially assume a preemptive algorithm for conflict resolution in which a conflicting transaction is aborted.

We now provide a high-level description of transaction execution with the location-based paradigm in order to provide background for the detailed description.

D. High-Level Description

A client wishing to perform transaction operations on replicated objects will behave as follows.

1) A client first submits a server? query to the closest location-server replica, and obtains a *hint* regarding the location of the *up-to-date replicas* of the desired object.
2) Clients use the hint to select any one of the up-to-date replicas as the leader and submit the operation to it.
3) Clients need only wait for the leader to respond and can then continue with the execution of the next transaction operation.
4) Each leader, after replying to the client, tries to inform enough other replicas of locks that need to be acquired and, in the case of a write, the operation to be executed. Thus, the nonleader replicas participate in parallel with the execution of the client.
5) During 2PC each leader ensures that each operation was performed on A_r or A_w replicas—otherwise the transaction is aborted.
6) If the client notices that the set of replicas participating in a write operation is different than the set in the hint provided by the extended location service, the client multicasts update-table requests to the location-server replicas.

From an availability point of view, the paradigm behaves fundamentally like any quorum-based replication technique, since as required by step 5), an operation can only succeed if at least A_r or A_w replicas have participated in its execution.

IV. Transaction Execution

We describe our replication method by examining the behavior of the main processes during transaction processing. Central to the method are the availability variables $A_r(x)$ and $A_w(x)$. These variables are related as follows: $A_r(x) + A_w(x) > n(x)$ and $2 \times A_w(x) > n(x)$, where $n(x)$ is the total number of replicas of x. In describing our method we assume familiarity with the two-phase commit protocol [17] and we concentrate on the additional features of our method.

A. Processing at the Client

Clients are responsible for finding the appropriate replica group for each object referenced by a transaction operation. A client sends the "server?" query to the leader 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 the leader replica for each object, to which the client will submit a request for executing the operation. The client will then wait for a reply and as soon as it is received the client will submit the next operation to its leader.

Clients are also the coordinators of the two-phase commit protocol. After a client has received the replies for all requests, 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 the commit phase is initiated. The client then broadcasts a commit message. Subsequently, the client sends an "update-table" message to the location server. This message contains a list identifying, for each object, the new up-to-date replicas. (As we shall see shortly, this message is only needed very rarely).

If any leader refuses to prepare, the transaction is aborted by broadcasting an abort message to all the leaders that communicated with the client. The leaders will propagate the commit/abort message to the other replicas. Fig. 2 details the client's behavior when executing transaction T.

```
LSR = location server replica closest to the client ;
send (LSR, "server?(x)", for all x referenced in the transaction T) ;

for each x referenced in T do
   up_to_date(x) = list of replicas of x returned by LSR ;

for each x referenced in T do begin
   leader(x) = the replica that is closest to the client ;
   send (leader(x), op(x)) ;
endfor ;
Leaders = { leader(x) for some x in T } ;
/* wait until leaders have responded to all requests */

/* The Two-Phase Commit Stage */
R-O(T) = true if T is a read-only transaction ;
bcast (Leaders, "prepare", T, R-O(T)) ;
/* receive replies */
for each x referenced in T do begin
   Vote[x] = prepare flag returned by leader(x);
   participating_sites(x) = the participants to a write operation on x ;
endfor ;
if there exists x referenced in T s.t. Vote[x] = "refuse" then begin
   bcast (Leaders, "abort", T) ;
   /* add recovery record to event history log */
   write (EHL, "aborted", T) ;
elseif not R-O(T) then
   write (EHL, "committed", T) ;
   for all x s.t. up_to_date(x) ≠ participating_sites(x) do
      add (x, participating_sites(x)) to update_table_list;
   Async_bcast (LSR, "update_table (update_table_list") ;
   bcast (Leaders, "commit", T) ;
endif ;
```

Fig. 2. The behavior of a client in the location-based paradigm.

B. 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 returned to the client. The propagation of the lock-acquisition requests (and in the case of a write operation, the operation request) occurs in the background.

After receiving a prepare message, the leader for a read (write) operation is responsible for verifying that $A_r(A_w)$ replicas have participated. The leader first broadcasts the prepare message with the operation request piggybacked onto it and sets a timeout waiting for responses. This broadcast is synchronous, requiring the leader to block until the timeout expires or until $A_r - 1$ (or all replicas) have replied to the read (write) operation. It is likely that, by this time, the required number of replicas has already received and acknowledged the operation requests. Thus, typically, the waiting time for the leader during the prepare phase will be small.

When and if the leader receives the required number of acknowledgments from the replicas, it replies with a "prepared" message to the client. This message, in the case of a write operation, will also contain the list of replicas that updated each object. If the timeout expires and fewer than $A_r - 1(A_w - 1)$ replicas replied then the leader refuses to prepare. Leaders for read operations also refuse to prepare if they notice that one of the participant replicas had a more recent version of the read object. (This happens, for example, when failures have caused the client to receive stale location information). Thus, stale read leaders never agree to prepare. Note that stale leaders for write operations need not refuse to prepare as long as they can propagate along with the updated object state the highest version number.

After receiving a "commit" message from the client, the leader will broadcast a "commit" message to the participant replicas, write the pair (transaction id, "committed") to the event history log, and wait until the participant replicas acknowledge the message. (Waiting is not required by some 2PC protocols [2]). At this point, the leader will release any pertinent locks and reply (if needed by the atomic commit protocol) with a "committed" message. Fig. 3 shows the algorithm executed by a leader R during the execution of a transaction T.

C. Processing at NonLeader Replicas

Nonleader replicas are involved in the processing of the transaction during the execution of the 2PC protocol and during asynchronous operation execution. When a nonleader replica receives an operation request, it obtains the required lock for the object. Subsequently, in the case of write operations, it will perform the indicated operation. It will acknowledge either the operation request 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"/abort" message from the leader it appends a (transaction id, "committed"/"aborted") 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 (if the atomic

commit protocol requires it). Fig. 4 depicts the behavior of a nonleader replica R during the execution of T.

D. Staleness of Location Information

As mentioned earlier, there exists the possibility that a transaction executes with stale location information. A location-server replica may have stale table entries because of failures which caused this replica to miss a number of update-table requests.

It is instructive to note that "update-table" commands occur as a result of some failure or recovery in the replica group for a replicated object. The reasonable assumption that failures are infrequent implies that "update-table" commands are infrequent. Furthermore, even if a location-server replica missed an update-table operation, its list of up-to-date sites for an object x will likely have many sites in common with the list of sites for x included in an "update-table" message. If the update-table message is sent as a result of a recovery, then no stale location information can cause the abortion of a transaction accessing x, since the stale location server will have a subset of the up-to-date replicas. If, however, the update-table message is sent because of some failures in x's group, then since a client may pick any replica as the leader from the list returned by the location server, it is likely that a client will indeed select a leader that has the most recent state for the referenced object. This holds even if the accessed location-server replica has missed a number of "update-table" messages. As an example, suppose an object x has five replicas and a failure causes an updating transaction to update only four of these. A subsequent client that obtained stale location information will select a proper leader with 80% probability. Suppose further that a subsequent transaction updated all five copies but another transaction obtained location information about x before the "update-table" message executed at the accessed location-server replica. Obviously, this transaction will not abort since it will select a leader from a subset of the set of sites that currently hold an up-to-date copy of x. These observations indicate that leaders for transaction operations will contain up-to-date versions of data objects, except for a very small number of cases.

Nonetheless, it is important to ensure correct transaction behavior in the presence of the above anomaly. Correctness is ensured by requiring each replica site to store a version number as part of the state of an object. Leaders for read operations receive the version numbers of the copies stored at the replica sites. If a leader for a read operation has a version number for the object that is smaller than the version number of some replica, then the transaction is aborted and restarted. Thus, transactions with leaders that read stale object states can never commit. This approach is not inefficient since from the above arguments one can conclude that restarts are very rare phenomena.

E. The Trade-Off Between Write and Update-Table Operations

As stated, our algorithm requires that a leader try to notify all other replicas of a write operation. In environments with many replicas and/or frequency write operations, this may cause considerable overhead. On the other hand, if only A_w replicas were asked to participate, then according to the stated algorithm, an update-table message will be needed for almost every update transaction. There are a number of solutions to this dilemma.

If the location server is not replicated, or has only a few replicas (fewer than the replicas of data objects) it may be more efficient to update only A_w replicas and send an update-table message at the end of the transaction. Alternatively, for each data object x a special group (of $A_w(x)$ replicas) can be defined. All leaders for write operations on x can now only send the operation to this group of replicas. For the vast majority of the time, the location-server replicas will list the special group as the up-to-date sites for x. When failures (and subsequent recoveries) occur, write operations will be performed on a different group (of $A_w(x)$ replicas) and thus update-table messages will be needed. These update-table messages will establish a new special group of A_w replicas. However, with this scheme, update-table messages will then be very rare, for the same reasons presented in the previous subsection. At the same time, write operations will only involve A_w replicas (except when failures and/or recoveries occur) and thus there will be only negligible overhead.

```
receive (client, op(x), T) ;
op_lock (x) ;
op (x) ;
send (client, x) ;
Async_bcast (replicas(x), op(x), T) ;
/* for reads, bcast to a group of A_r(x)-1 replicas suffices */

receive (client, "prepare", T, R-O(T)) ;
bcast (replicas(x), "op(x); prepare", T, R-O(T)) ;
if op(x) = r(x) then wait (for timeout or A_r(x) replies) ;
if op(x) = w(x) then wait (for timeout or replies from all replicas) ;
participating_sites(x) = replicas of x that replied to the leader ;
for all R1∈ participating_sites(x) do
   version_R1(x) = the version number of x sent by replica R1 ;

if card(participating_sites(x)) < A_op(x) or
   (op(x) = read(x) and there exists R1 s.t. version_R1(x) > version_R(x))
then send (client, "refuse", T) ;
else
   write (EHL, "prepared", T) ;
   send (client, "prepared", T, participating_sites(x)) ;
endif ;

/* The second phase of 2PC is needed only for update transactions */
if R-O(T) = false then begin
   receive (client, code, T) ;
   if code = "abort" then begin
      bcast (participating_sites(x), "abort", T) ;
      write (EHL, "aborted", T) ;
      undo (T);
   else
      max_version = max (version_R1(x), R1∈ participating_sites(x)) + 1 ;
      bcast (participating_sites(x), "commit", T, max_version) ;
      write (EHL, "committed", T) ;
   endif ;
endif ;
release_locks (x) ;
```

Fig. 3. A leader's (R) behavior in the location-based paradigm.

```
/* Operation execution */
receive (leader(x), op(x), T) ;
op_lock (x) ;
if op(x) = w(x) then op (x) ;
send (leader(x), version_R(x)) ;

/* Prepare phase of 2PC */
receive (leader(x), "op(x); prepare", T, R-O(T)) ;
if op(x) has not been performed yet then begin
   op_lock (x) ;
   if op(x) = w(x) then op (x) ;
endif
write (EHL, "prepared", T) ;
send (leader(x), "prepared", T) ;

/* The second phase of 2PC is needed only for update transactions */
if R-O(T) = false then begin
   receive (leader(x), code, T, max_version) ;
   if code = "abort" then begin
      write (EHL, "aborted", T) ;
      send (leader(x), "aborted", T) ;
      undo (T);
   else
      version_R(x) = max_version ;
      send (leader(x), "committed", T) ;
      write (EHL, "committed", T) ;
   endif ;
endif ;
release_locks (x) ;
```

Fig. 4. Nonleader replica's (R) behavior in the location-based paradigm.

V. Correctness Proof

The formal model and proof technique are taken from [3]. We briefly introduce serializability theory for nonreplicated systems and, subsequently, the extensions made by Bernstein and Goodman to obtain the theory for replicated systems.

A. Correctness of NonReplicated 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 $T_i = (\sum_i, <_i)$, with $\sum_i$ representing the operations in transaction i and $<_i$ depicting the order in which these operations are performed. A log over a set of transactions $T = \{T_0, T_1, \ldots, T_n\}$ is modeled by a poset L with the following properties: 1) $L = (\sum, <)$, with $\sum = \cup_{i=0}^{n} \sum_i$; 2) $< \supseteq \cup_{i=0}^{n} <_i$; 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 logs are said to be equivalent if they have the same reads-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 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 $E = \{T_i \rightarrow T_j |$ there exist conflicting operations $\mathrm{op}_i, \mathrm{op}_j$ with $\mathrm{op}_i < \mathrm{op}_j\}$.

Theorem 1: A concurrency-control algorithm for nonreplicated systems satisfies the serializability correctness criterion if $SG(L)$ is acyclic [3, 24].

B. Correctness in Replicated Systems

The serializability theory for nonreplicated systems 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 $x_{a_1}, x_{a_2}, \ldots, x_{a_t}$, 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: $t(r_i(x)) = \{r_i(x_a)\}$, where in addition to the replica that is read, a total of n copies are (read-locked) accessed, where n is enough to guarantee correctness. $t(w_i(x)) = \{w_i(x_1), w_i(x_2), \ldots, w_i(x_m)\}$, where m copies are enough to guarantee correctness.

A replicated data log (rd log) over a set of transactions is a poset L with the following properties: 1) $L = (\sum, <)$, with $\sum = t(\cup_{i=0}^{n} \sum_i)$; 2) for each T_i and $\mathrm{op}_{i1}, \mathrm{op}_{i2}$ if $\mathrm{op}_{i2} < \mathrm{op}_{i2}$ then each physical operation in $t(\mathrm{op}_{i1})$ is $<$-related to $t(\mathrm{op}_{i2})$; 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 s serial *one-copy* log. The other definitions remain as before. The one-copy serialization graph for an rd log L, 1-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-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: For an rd log L, if 1-SG(L) is acyclic then L is one-copy serializable [3].

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

C. The Proof

The proof requires four basic steps: 1) formalize the behavior 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$_i(x_a) \rightarrow r_i(x_a) \rightarrow \{rd\text{-lock}_i(x_1), \ldots, rd\text{-lock}_i(x_m)\}$, where $A_r(x)-1 \leq m \leq n-1$, 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$_k(x_a) \rightarrow w_k(x_a) \rightarrow \{(\text{wr-lock}_k(x_1) \rightarrow w_k(x_1)), \ldots, (\text{wr-lock}_k(x_m) \rightarrow w_k(x_m))\}$, with $A_w(x) - 1 \leq m \leq n - 1$ and n 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 lemmas play a central role in the proof.

Lemma 1: For any committed transaction T_i, any leader l_i for an operation $r_i(x), r_i(x) \in T_i$, contained the most recent version for x.

Proof: Suppose to the contrary that T_i committed and that one of its leaders, l_i read a stale copy of x. There must exist a committed transaction T_j that caused the "update-table" message that made the location information received by T_i obsolete. Denote the leader for $w_j(x)$ (the write operation that triggered the "update-table" command) l_j. $v_i(x)$ and $v_j(x)$ represent the version of x that l_i read and the version that $w_j(x)$ installed.

From the description of our method we know that since T_j committed, all of its leaders (including l_j) agreed to prepare during 2PC. This can only happen if at least $A_w(x)$ replicas for object x obtained write locks and executed $w_j(x)$, installing $v_j(x)$. Since we assumed that l_i read a stale copy it must hold that $v_i(x) < v_j(x)$. Because T_i commits we conclude that $A_r(x)$ replicas agreed to prepare during 2PC. But this can only happen if none of these replicas had a version number that was greater than or equal to $v_i(x)$. Since $A_r(x) + A_w(x) > n(x)$ we must have that at least one of the replicas of x, say r, had installed $v_j(x)$ and received a prepare message with $v_i(x)$ from l_i. But in this case the leader replica will refuse to prepare because $v_i(x) < v_j(x)$. This will cause the abortion of T_i, contradicting our assumption that T_i committed having performed some operation on a stale copy. Hence there cannot exist such a transaction T_j.

Therefore, we conclude that any committed transaction read up-to-date object states for all of its objects and thus Lemma 1 holds. □

Using the above lemma and noting that serialization graphs model the effects of committed transactions we develop the following two lemmas to show the acyclicity of 1-SG(L).

Lemma 2: 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 $\text{op}_i, \text{op}_j \in L$, such that $\text{op}_i < \text{op}_j$ 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. Since locks are held until the commitment of transactions, we can conclude that T_j acquired the locks(s) necessary for performing op_j after T_i committed. Thus T_i committed before T_j. □

Lemma 3: The serialization graph $SG(L)$ is exactly the same as the one-copy serialization graph, 1-$SG(L)$.

Proof: To construct 1-$SG(L)$ we first construct $SG(L)$ and the add enough edges so that i) there exists in 1-$SG(L)$ 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 $T_i \ll_x T_k$ there is 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. Our method requires that logical writes be physically applied to at least $A_w(x)$ of the copies of x. 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 $w_j(x_a) \in L$, since $2 \times A_w(x) > n(x)$. 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 is <-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 $SG(L)$ 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 will have been obtained at $A_r(x)$ replicas (by the end of prepare phase) for every read operation. Write locks will also have been obtained at $A_w(x)$ replicas. Recall that rd-lock operations conflict with wr-lock operations, provided they are applied on (at least) one common copy. Since $A_r(x) + A_w(x) > n(x)$, 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 2 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 (from Lemma 1). Thus, assuming $w_k(x_a) < r_j(x_a)$ leads to a contradiction. 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 3 holds. □

The main theorem follows directly from these lemmas.

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

Proof: (by contradiction)

Lemma 3 implies that the statement of Lemma 2 holds also for 1-$SG(L)$. Assume that 1-$SG(L)$ contains a cycle, say, $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_r \rightarrow T_1$. From Lemma 2 we know that T_1 commits before $T_2, \ldots,$ 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 incorrect.

VI. Site Crashes and Recoveries

In the first part of this section we examine the effect of site crashes on the protocol. Our aim is to show that the location-based method deals efficiently with such failure in terms of service disruption and in terms of the number of messages required to deal with such failures. Subsequently we address crash recovery.

A. Crashes

The crashing of nodes running location-server replicas can introduced a two-round message delay and two more message (i.e., the additional "server?" query message and the message responding to it).

We now examine in detail the crashing of a selected leader site before and after the operation is executed (cases 1 and 2 respectively), and the crashing of a nonleader that executed an operation but failed before receiving the prepare message (case 3).

1) A Client Selects as a Leader a Failed Site: The client will timeout waiting for a response to its request. Then it can simply select another leader by choosing another replica from the list of up-to-date sites that it received from the extended location service. Note that crashing in this case introduces a two-round message delay (one round to send the operation to the leader and another round for the timeout expiration). There is only one additional message.

2) A Nonleader Replica That Participated in the Execution of an Operation Fails Before 2PC: In this case the location-based method suffers from no overhead (unless the operation is a read operation and the leader tried to involve only A_r-1 other replicas, in which case the leader must find a new replica to replace the failed one).

3) The Leader Crashes After Executing the Operation: In this case there will be no reply to the client's prepare message. The client then can play the role the leader replica plays during 2PC by broadcasting the message "prepare; op(x)" as the leader would have done. Since the client has received the leader's version of the object it can easily check if a stale object was read and also ensure that enough replicas participated in the operation. In this case it can proceed as if the leader were alive and had replied "prepared". Therefore, the additional cost is a two-round delay to notice the leader's crash. No additional messages are required since the prepare message must be broadcast in any event.

B. Crash Recovery

Transaction-based distributed systems must employ stable-storage logs in order to recover from site crashes and media failures. (We assume that such logs are as defined in [20] i.e., implemented on mirrored disks).

Newly-recovered replica sites in the location-based paradigm do not need to execute a multi-site recovery protocol to bring their copies up-to-date and update objects accessed by uncertain transactions. The local event-history log (which is implemented on disk) can be used to make the contents of the database consistent for transactions outside the uncertainty period. In other words, the local event history log is used to install the changes of committed transactions to the permanent store and perhaps to undo any effects of aborted or nonprepared transactions. To recover from media failures a replica may use the event-history logs at the other replicas or may implement the event history log in stable storage (i.e., in mirrored disks [20]).

The "uncertain" transactions in the log—i.e., those with a "prepared" and no "committed" or "aborted" entry in the local log—are treated differently. For all uncertain objects (i.e., those updated by uncertain transactions) the "before images" will be returned to all subsequent requests. This implies that it is possible that a leader agreed to prepare receiving exactly $A_w(x)$ replies for a $w(x)$ operation and because of a crash failure one of the replicas that replied to the leader will be using the version of x prior to the $w(x)$ operation. The end result is the same as if the $w(x)$ was in effect committed at fewer than $A_w(x)$ replicas.

We ensure that the above scenario does not lead to serialization problems as follows. A recovering replica marks all of its uncertain objects with an "uncertain flag". Afterwards, the replica is ready to receive requests and thus it is immediately reintroduced into the system. Each object is unmarked as soon as the first write operation executes (and commits) at the local site. In addition, the recovered replica can participate in operations but with these additional restrictions:

1) Along with the state of a flagged object, the replica returns the associated flag.
2) A client must wait for only one response to its read request, but the response must not be flagged.
3) A leader for an operation, in order to agree to prepare during 2PC, must have gathered at least $A_r(x)$ ($A_w(x)$) *nonflagged* responses for a read (write) operation.

For objects that are heavily read-intensive (i.e., write operations are uncommon) a leader may also respond to replicas which sent flagged responses by propagating its version of the object's state if it notices that the replica's state is stale. This can be done after the leader has agreed to prepare, and

asynchronously with respect to the execution of the on-going transaction. If a leader notices that a flagged response was not stale then when it forwards the commit message it can also indicate to the replica that it should unmark its state.

It is important to discuss why this crash-recovery algorithm does not result in nonserializable executions and does not interfere with the achievement of our performance goals. First, note that the correctness proof of Section V still applies. Assume there are f flagged (and $N-f$ nonflagged) replicas. A read (write) leader must collect $A_r(A_w)$ nonflagged responses. There are $A_w - f$ up-to-date, nonflagged replicas. But $A_w - f + A_w > N - f$ since $2 \times A_w > N$. Thus, a write leader will always access at least one up-to-date nonflagged replica. Similarly, $A_w - f + A_r > N - f$ and thus a read leader will also always access an up-to-date, nonflagged replica.

Second, most clients would still wait for a single reply from the leader replica. This is so because all object accesses are filtered through the extended location service. Thus in most cases, while a replica was unavailable, write operations will have created new entries in the location tables listing the new sets of up-to-date sites of objects with copies at the failed replica site. Therefore, subsequent clients will be advised to choose as a leader another replica.

The advantages of the above crash-recovery algorithm are significant. First, independent crash recovery from site crashes is facilitated. Independent recovery is very desirable because it eliminates the overhead caused by messages which are needed between the recovering replica and the other replicas to resolve the replica's uncertainty and/or bring the local data objects up-to-date. Furthermore, independent recovery avoids the wasting of resources since, for example, locks on objects written by "uncertain" transactions are released immediately and operations can execute on them without waiting for the resolution of the uncertainty. Second, the recovering replica is reintroduced into the system very rapidly and this will have an effect on both availability and performance (for nearby clients).

Crash recovery of nodes with location-server replicas is inexpensive. A recovering node which has been down for a short time may simply use its disk-based location table to continue processing "server?" queries and "update-table" operations. This decision is based on our earlier discussion that established the rarity of location-information staleness and, furthermore, the rarity of transactions accessing stale leaders even in the case that clients receive stale location information. If a newly recovered location-server replica has been down for a long time, it may refrain from replying to clients. Alternatively, when it replies to clients it can remove any local replica (which will be stale with a high probability) from the list of up-to-date replicas. This will avoid the problem of newly-recovered stale location-server replicas always suggesting their own copies as leaders when replying to nearby clients.

VII. The Performance of the Location-Based Paradigm

In order to substantiate our performance claims, we have undertaken a simulation study which focuses on the transaction-execution-delay of replication protocols. We have constructed, along with a model of the location-based protocol, two generic models reflecting different approaches to the design of replication-control protocols. These represent the primary-site paradigm for replication and quorum-based protocols. In addition, we have constructed a model for nonreplicated systems.

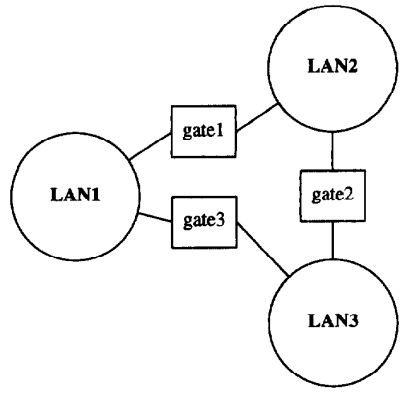

Fig. 5. The interconnection of the simulated distributed system's components.

A. The Environment

The simulated distributed system consists of a number of local-area networks which are connected by gateways. This type of configuration is commonly found in today's distributed systems. Fig. 5 depicts the particular configuration used in this simulation study.

Each LAN component consists of nine sites. Each site has one cpu and one disk where all its objects are stored. Objects stored at a particular site are said to be local to the processes running at that site and semi-local to processes running at sites which are in the same LAN component. Also, they are said to be remote to all processes running at sites in different LAN components. The concepts of local, semi-local and remote operations are defined similarly.

B. The Cost Model

Hereafter "cost of an operation" will refer to the delay incurred when executing this operation. The cost of operation execution consists of three components: *system_cost*, *transmit_cost*, and *local_op*. *system_cost* represents the (cpu) cost attributed to executing system calls at the sender and receiver. *transmit_cost* represents the actual transmission time for sending messages over a network. *local_op* represents the cost of locally reading or writing a data object and acquiring the proper lock (thus it involves the cost to access the disk and the cost to access main-memory concurrency control information).

The cost of an operation therefore is the sum of the above three costs. Local operations, of course, do not have a *transmit_cost* component. Operation costs depend on the relative location of the sender and receiver and on the number

of required acknowledgments. If an operation requires passing through a gateway (i.e., a remote operation) the cpu and transmit_cost components of the operation cost are modelled as twice the corresponding costs of a semi-local operation. (This estimation can be explained by viewing a remote operation as requiring twice as many messages, with twice as many senders and receivers: sender—gateway—receiver.)

The transmission cost for an operation, which requires multicasting to n replicas and waiting for m replies, is modelled as m times the *transmit_cost* of an operation using unicast. This model for the multicasting cost can be easily justified from performance studies of multicast protocols which show that the (end-to-end) latency cost increases linearly with the number of participants (i.e., members of the destination group) [4].

C. The Protocols

We define the generic protocol of *synchronous replicated operation execution* (SROE) as one in which a client, before proceeding with the next transaction operation, blocks until a majority of replicas have replied to its previous request. This model is representative of many well-known quorum-based replication-control protocols in the literature [7], [9], [12], [14], [25].

In the model of the primary-copy (PC) paradigm for replication one replica of each object is designated as the primary and is responsible for servicing all requests for operations directed to the replicated object. Clients must first determine which is the primary replica and submit the operation request to it. Since only primaries perform operations, clients must wait for only the primary replica to respond before issuing their next operation request. The same type of behavior characterizes the model for nonreplicated systems (NRS), where, instead of a primary, clients determine the location of and contact the single copy of a referenced object.

The model for the location-based replication method (LBR) faithfully depicts the behavior of the protocol as presented in Section IV. A client need only wait for a response from the leader before proceeding with its next operation.

The cost of executing a transaction consists of two components: one attributed to the execution of transaction operations and another attributed to executing the two-phase commit protocol. Above we have discussed the cost of local, semi-local, and remote operations, and operations requiring multicasting. However, note that although LBR, PC, and NRS all require clients to wait for only one replica, operation costs will in general differ. For example, if an object has three replicas, one in each LAN, under LBR the operation will be either local or semi-local; under PC and NRS the operation can also be remote (with probability 66.7%).

The protocols incur different costs for atomic commitment. The difference primarily results from the requirement that nonreplicated systems use stable-storage recovery logs and from the different number of replicas that have to be contacted by a leader (in LBR), a primary replica (in PC), or a client (in SROE) before it can decide whether to prepare. The cost of atomic commitment is given by the cost of executing the first phase of 2PC since it is only after the end of the first phase that a transaction can be considered committed.

The cost of atomic commitment in LBR consists of the cost for communicating the prepare message to the leaders, each of which then has to relay the message to the nonleader replicas. Thus it is composed of two multicasts (one from the client to the leaders and one from the leaders to the nonleader replicas). There is also the cost of two disk writes, one at the client and one at each replica (replicas do the writes in parallel) for the forcing of the commit/abort record at the client and the "prepared" record at the replicas.

With regard to the cost for atomic commitment in NRS, we assume that the additional cost of writing to stable storage (required by NRS) is the same as that of two-round communication in LBR between the leader and the other replicas during the prepare phase. Thus, the LBR and NRS protocols are assumed to have the same cost for atomic commitment.

The PC protocol implemented in our simulation is very similar to that of viewstamped replication [22]. In this protocol during the prepare phase each primary need wait only for a majority of responses (instead of responses from all available replicas as in the LBR protocol). Therefore, the atomic commitment cost for PC is given by two multicasts: one from the client to the primaries and one from each primary to the other replicas. Note, however, that the second multicast in PC will be cheaper than the second multicast in LBR since primaries wait for fewer replicas than leaders. (Recall the discussion for calculating the costs of operations requiring multicasting). Like LBR, there is also the cost for two disk writes for commit/abort and prepare log records in the client's and replicas' logs respectively.

Finally, the cost for atomic commitment in the SROE protocol is modelled as being the same as the cost for atomic commitment in the PC protocol since exactly the same number of replicas (i.e., a majority) must be contacted by the client during 2PC in both protocols.

D. Simulation Objectives and Parameters

As mentioned above, the primary objective of this study is to substantiate the claim of decreased transaction-execution delay in the location-based protocol. Given the characteristics of this protocol the simulation study attempts to

- Verify and quantify the improvements in transaction-execution delay.
- Study the effects of concurrency conflicts on the performance of the location-based method.
- Investigate the effect of transaction size on the performance improvement.
- Determine the validity of our assumption that stale-read restarts are rare events.

For fairness reasons we implemented a classic strict 2PL algorithm for concurrency control for all tested protocols. Thus, conflicting transactions wait in lock queues. Given this blocking behavior, we wanted to investigate how concurrency conflicts affectthe relative performance of the protocols.

Similarly, recall that in LBR a leader must wait for all (available) replicas to respond to its prepare message during 2PC, whereas in PC and SROE only majorities are needed. Therefore, the effect of the size of transactions on the relative

performance of the protocols becomes significant since the cost of 2PC will be a significant contributor to the overall cost of small transactions and may offset the latency benefits of LBR for small transactions.

Stale read leaders cause restarts and thus may affect the performance of LBR. In addition, it is important to investigate if the benefits of the proposed recovery algorithm are offset by additional delays during normal transaction processing. Thus our simulation collects statistics regarding the number of stale-read restarts witnessed during normal transaction execution. This information can prove that the proposed crash recovery algorithm does not result in a large number of abortions and restarts.

1) Parameter Estimation and Other Values: To facilitate the investigation of the effect of concurrency conflicts on the performance of the location-based method we employ the concept of *access intensity* which consists of two parameters: *object_intensity* and *operation_intensity*. Its purpose is to model the nonuniform distribution of operation requests to data objects that characterizes real-world data-access patterns. In effect, *object_intensity* partitions the set of all data objects into two subsets: a subset consisting of intensively-accessed objects and another consisting of less-frequently accessed objects. The *operation_intensity* parameter controls the fraction of transaction operations that are directed to access-intensive objects. For example, if *object_intensity* = .1 and *operation_intensity* = .8 then 80% of the operations of each transaction are directed to the access-intensive data objects which constitute 10% of the total object set. We assume that within each subset accesses are uniformly distributed across objects. To observe the effect of conflicts we set the pair of (*object_intensity, operation_intensity*) to the values (.01, .8), (0.25, .8) and (.05, .8).

We use the *size_of_t* parameter to investigate the effects of transaction size on the performance of the protocols. We believe that for most transaction systems in which one-copy serializability is adopted as the correctness criterion, transactions will (and must) be small. Since our method is intended for such systems, we chose the values for *size_of_t* from $\{3, 6, 12\}$.

To compute transaction costs we need to assign values to the variables *system_cost, transmit_cost*, and *local_op*. The system cost appears to be significant [6, 11, 15], accounting for at least half of the total cost for a semi-local operation. The actual values depend, of course, on the processor speed and the communication bandwidth. The following values appear to be appropriate; *system_cost* = 6, *local_op* = 20.5, and *transmit_cost* = 2 × 2.5 milliseconds. For *local_op* 20 ms is attributed to accessing the disk and 0.5 ms attributed to maintaining lock information.

In order to keep the number of experiments manageable we decided to use a constant mean transaction interarrival rate of 90 tps. Furthermore, in order to keep the size of each experiment manageable, instead of allowing 90 tps per site of the system we introduce transactions at 90 tps rates for each LAN component. Therefore, the transaction arrival rate was set to 270 tps for the entire system.

Another important value for our simulation is the number of data objects that exist in our system. This is harder to estimate. We estimated the number of data objects in a typical system to be 100,000. (Data objects actually refer to lock granules). To be consistent with the previously reported estimates of transaction interarrival times we assume the existence of 100,000 objects per LAN component instead of per system site.

Again, in order to keep the number of experiments manageable we used a constant number of replicas, which for all objects was set to five. This number is justified since, given the system environment, most objects then become semi_local which implies improved availability and performance in the presence of partitionings—thus satisfying adequately both motivations for replication.

Finally, the read-to-write ratio was fixed at 4:1.

E. Transaction Execution

Transaction arrival events are assumed to be *poisson*. They occur with a mean of 270 transaction per second. Transaction origination points are uniformly distributed across all sites of the system. Replica sites for each object are chosen randomly with all sites being equally probable as storing sites for a replica of the object.

Clients are assumed to have zero-cost access to location information. Obtaining location information is not considered for any of the simulated protocols. For instance, the cost of determining the up-to-date replicas from which a leader will be chosen in the location-based protocol, the primary of a replicated data object in the primary-copy protocol, or simply the replica group in SROE is not computed. This was done mainly for simplicity reasons since it would be hard to model features such as caching location information and maintaining cache consistency.

When a transaction arrives, the process of executing it begins immediately. The next operation (i.e., the operation type and the object to be accessed) is determined using the read-to-write ratio, and the object and operation intensities. Next, an appropriate replica (or set of replicas for the SROE model) is selected. Thus, a leader, the primary, the single site storing a copy of the referenced object (in the case of a nonreplicated system), or a quorum of sites (in the SROE model) is located. Given this information the client then submits the operation request. The arrival time of the request at the replica(s) is determined using the model discussed earlier (i.e., depending on the relative positions of sender and receiver, the number of participants, etc) and the request is placed in the appropriate job queue to be processed at that time. Thus, the arrival time will be equal to half of the total cost for performing the operation, without including the local_op cost.

When the time comes to process a request, it is served by the replica and a reply is sent to the client. As mentioned, a lock queue is associated with each replica. Conflicting transactions are put at the tail of the lock queue. When the lock-owner transaction commits/aborts, locks are released and eligible transactions at the head of the lock queues (one writer or many readers) are scheduled. The arrival time of the reply is computed in the same way as above, except that it also includes the local_op cost. The reply is placed in the client's queue, to

be processed at that time. In the LBR and PC models the leader and the primary relay the request to the other replicas using multicasting and observing the associated cost.

This process continues for each operation. When the client has received the response(s) to the last operation it initiates the 2PC protocol. The end of the first phase occurs at current time plus a time equal to the cost of atomic commitment (recall that the cost of atomic commitment does not include the second phase of 2PC). At the end of the first phase for all protocols the client will check if enough replicas have participated in the prepare phase. If a majority of replicas did not participate in any operation, the transaction will be aborted. The lock-release time is computed as the average time it would take the commit or abort message to arrive at a replica. The lock-release event is thus scheduled at that time.

Finally, if an update-table message is needed in LBR, this message is sent and will arrive at all *accessible* location-server replicas at the same time that the replicas will receive the lock-release message.

F. The Results

The results reported herein address the previously-mentioned objectives of the simulation study. In addition, they provide the justification for some of the choices we have made in designing the experiments.

Independent determination of sample mean values through the replication of runs with different random number sequences was used to gain confidence in the simulation output [10]. The reported results (unless explicitly stated otherwise) have a 90% confidence interval of less than a few percent of each reported value.

The results reported in the first two experiments [Sections VII-E1) and VII-E2)] reflect the performance of the system when no failures occur. From other studies (not reported here - see [28]) we have found that the effect of failures on transaction-execution delay was negligible.

1) The Impact of Concurrency Conflicts: We elected to use LBR with a concurrency control algorithm which requires transactions to wait for conflicting locks (strict 2PL). This can potentially have negative effects on the performance of LBR. For example, if transaction $T1$ tries to write x the same time as $T2$ tries to read x, the following scenario is possible: $T1$ acquires write-locks on three replicas and $T2$ acquires read locks on two replicas. $T2$ will be blocked until $T1$ commits/aborts. After $T2$ acquires a lock on one of the replicas updated by $T1$, its leader for x will be informed of a newer version of x and thus $T2$ will be aborted. This is a stale-read restart. We must therefore test whether LBR's performance can be significantly degraded as a result of concurrency conflicts. (Note that if we had implemented a concurrency control algorithm which restarts a conflicting transaction, the above restart would also occur. Thus, the simulated LBR is general enough.

In Fig. 6 we present the relative performance of the protocols under varying concurrency-conflict rates.

The figure shows clearly that the location-based protocol outperforms the other protocols under all tested access-intensity pair values. The following table quantifies in more detail the relative-performance information of Fig. 6.

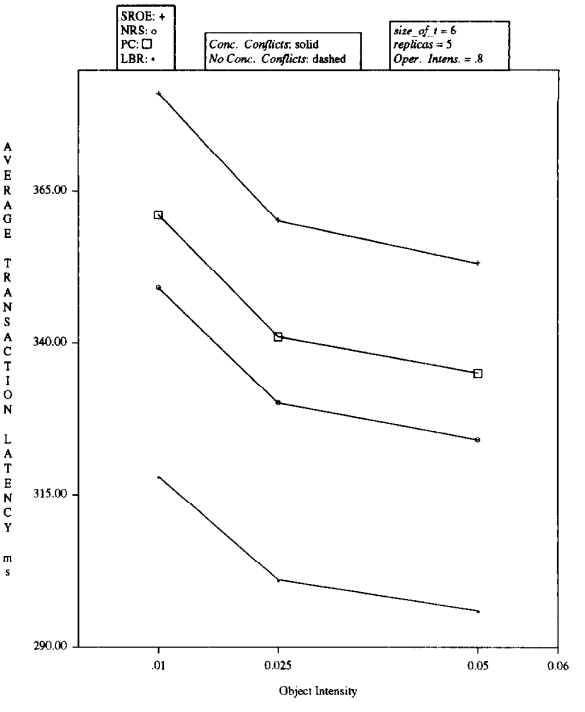

Fig. 6. Execution delay with varying concurrency-conflict rates.

From this information we can see that the performance difference remains constant over the different access-intensity pair values. This can be partly attributed to choosing the same concurrency control algorithm for all protocols. The central conclusion is that concurrency conflicts do not offset the comparative savings achieved by LBR. In fact using a concurrency control algorithm which requires waiting for locks (although in theory it introduces more stale-read restarts) does not have any significant negative impact on the performance of LBR since LBR consistently outperforms the other protocols by the same margin, regardless of the conflict rates.

2) The Impact of Transaction Size: In Fig. 7 we present the results of the study on the effect of transaction size on the relative performance of the protocols. In order to isolate the effects of concurrency we fixed the access intensity pair value at (.05, .8).

Fig. 7 clearly shows the superiority of LBR over all other protocols for all studied transaction sizes. In order to quantify further the improvement in performance we present Table II which shows exactly the performance penalty associated with each protocol for the given transaction sizes.

The central conclusion is that the performance penalties associated with PC, NRS, and SROE consistently increase, in both relative and absolute terms, as the transaction size increases. Thus, even with larger transactions, we can expect LBR to perform better. For larger transactions, the additional overhead incurred by LBR during 2PC is easily offset by the benefits of performing operation at leaders.

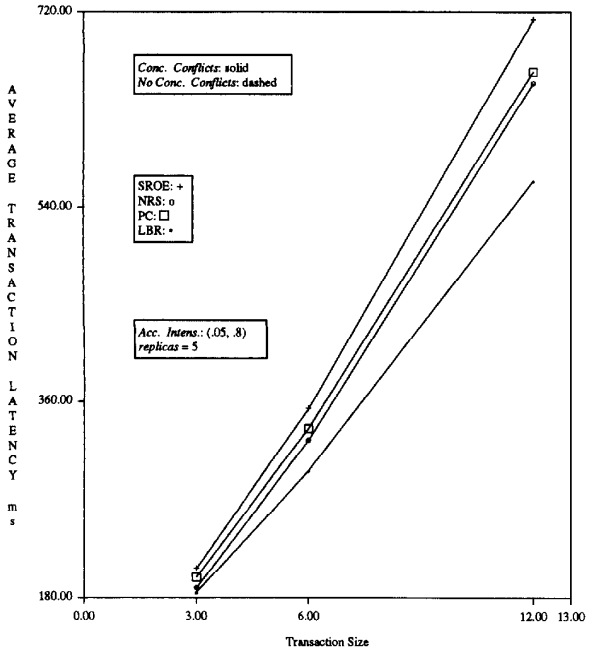

Fig. 7. Execution delay under varying transaction sizes.

TABLE I
PERFORMANCE PENALTIES UNDER VARYING CONCURRENCY CONFLICTS

Relative Performance Penalty of PC, NRS, and SROE				
Intensity	LBR	PC	NRS	SROE
(.01, .8)	318	13.5% (361)	9.7% (349)	19.8% (381)
(.025, .8)	301	13.3% (341)	9.6% (330)	19.6% (360)
(.05, .8)	296	13.2% (335)	9.5% (324)	19.3% (353)

TABLE II
THE RELATIVE PERFORMANCE PENALTIES FOR VARYING TRANSACTION SIZES

Relative Performance Penalty for Varying Transaction Sizes				
Transaction Size	LBR	PC	NRS	SROE
3	185	7.6% (199)	2.2% (189)	11.9% (207)
6	296	13.2% (335)	9.5% (324)	19.3% (353)
12	562	18.3% (665)	16.4% (654)	26.9% (713)

3) The Impact of Stale Leaders: When read operations are directed to stale leaders (e.g., replicas that are chosen to be leaders as a result of stale location information) they will cause the abortion and restart of the transaction. Thus, it is worth investigating if, in the presence of failures, restarts caused by stale location information introduce significant additional delays.

Additionally, in the description of the crash-recovery algorithm the claim was made that the location-based paradigm can afford not to have recovering replica sites update their replicas. This will certainly reduce communication significantly since many replicas could be out-of-date. However, if this approach to recovery caused many transactions to be aborted one could claim that there are no real savings but simply a delay of the overhead. Therefore, investigating the number of stale-leader restarts can also show that the proposed crash-recovery algorithm indeed reduces costs for the distributed system.

The study we conducted for measuring stale-leader restarts involved running a number of experiments while varying the availability of each site. Given the level of detail of the above simulation program and realistic values for the mean failure and recovery rates it would be extremely time consuming to perform this study. Therefore, we decided to design a new simulation program. The events of primary interest in this program are failures, recoveries and read/write operations. In order to test our simulated system with a large number of failures, read and write operations were assumed to occur immediately. In other words, we did not model details such as sending/receiving messages, concurrency conflicts, atomic commitment, etc. Instead, to process a write operation, the sites storing replicas were checked for accessibility and all accessible replicas were immediately updated with the new version. If any update-table messages were needed, these were sent to the available location-server replicas. To process a read operation, a leader was chosen as before using the extended location service. During the execution of read operations, the program was also checking all nonleader replicas. If any nonleader had a version number greater than the leader's version, the *number_of_stale_reads* variable was incremented. Thus, this simple model captures accurately the behavior of LBR with respect to the *number_of_stale_reads* variable.

The simulated system remains as before. All parameter values remain the same, except for *size_of_1* which was set to ten. We also assumed that in each LAN a total of 90 transactions (i.e., 270 transaction and 2700 operations in total) were processed per second. The access-intensity pair value was set to (0.01, .80). Finally, we also varied the availability of each site (i.e., the percentage of time during which sites are operational) to investigate how the number of stale-leader restarts is affected. The mean-time-to-failure parameter was set at 10 seconds. The mean-time-to-recovery parameter was varied to model site availabilities of 70%, 80%, 90%, and 95%. Also, the values of the mean-time-to-recovery parameter, were large enough so that, while a replica site was nonoperational, all of the access intensive objects were updated at least once. In effect, this created a worst-case scenario for LBR, since any transactions using a newly-recovered site as a read leader for an access-intensive object will be aborted.

The reported results have a 90% confidence interval of ± 0.27, ± 0.31, ± 0.18, and ± 0.05 for site availabilities of 70%, 80%, 90%, and 95%, respectively. In Fig. 8 we show the results of the study.

Detailed information, such as the average number of failures, the average number (and percentage) of stale read operations, and the average of the total number of read operations, is presented in Table III.

For the different site availability values the number of stale reads ranged from 0.33 to 1.32 percent of all read operations. We note that in a real-world system the above numbers would be even smaller. This is because there will be a much larger number of operations which execute between failures. On the other hand, the number of stale reads will remain at similar levels (recall that the recovery rates in our experiments were

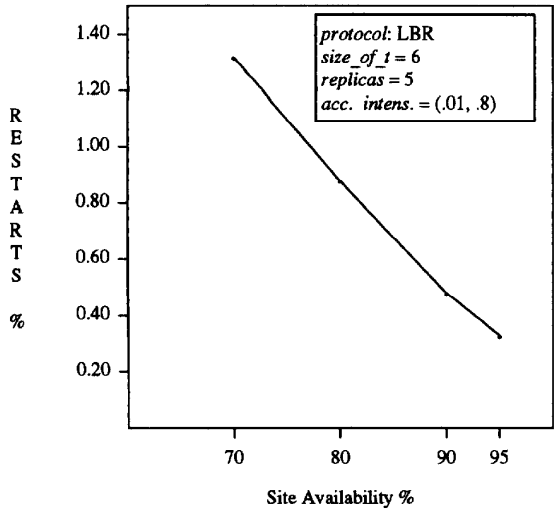

Fig. 8. Stale-leader restarts under varying site availabilities.

selected so that 100% of the access-intensive objects will have been updated before a replica recovers).

The conclusion therefore is that restarts caused from reading stale leader replicas do not affect the latency of transactions in the location-based paradigm. Furthermore, such a small number of restarts over so many failure events clearly shows that the crash-recovery algorithm of the location-based paradigm indeed results in considerable savings since no communication is needed between the recovering site and the other replica sites to update its stale objects.

The above conclusion is also intuitively reasonable. The number of restarts caused by the recovery algorithm in LBR should be independent of the number of objects that have been updated while a replica site was down. This is so since all client accesses to data objects are directed through the location server. For all stale objects the location server will have a list of up-to-date sites. Therefore, a newly recovered site with stale objects will not be asked to be the leader, except in very few cases.

VIII. Comparison With Other Work

In this section we concentrate on the performance and availability characteristics of the location-based method. We also qualitatively compare this method to other well-known methods.

A. Performance Considerations

Our goal was to minimize the cost of replication in distributed systems. This partly translates to minimizing transaction-execution delay. Naturally, therefore, we attempted to achieve transaction-execution delays similar to those occurring in one-copy systems. The only prior method that has achieved this goal is the viewstamped method [22].

Before a transaction starts executing, the client requests location service. In a nonreplicated 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. Our paradigm can also cache extended location information, given the rarity with which this information changes and the rarity of restarts induced by stale location information.

TABLE III
Details of the Stale-Leader Restarts Study

Stale-leader Restarts Experiments			
Availability	Failures	Stale Reads (%)	Reads
70%	99	28649 (1.32)	2,165,374
80%	97	19223 (0.88)	2,175,058
90%	99	10430 (0.48)	2,181,341
95%	102	7167 (0.33)	2,181,708

After obtaining the necessary location information, the client submits each operation to an appropriate site that can execute it. In nonreplicated 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. Note, though, that in the location-based method a client can select a leader that is local, if possible, or at least nearby. In this sense, our concept of a leader can help improve performance. Note that this performance benefit is not possible in nonreplicated systems or in primary-copy protocols. Since a client is involved in three phases of message exchange with each primary (for executing the operation and participating in 2PC) distant primaries may significantly increase the response time.

When all transaction operations have been performed, the 2PC protocol is initiated. A site participating in 2PC in a nonreplicated 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 logs. In both the location-based and viewstamped methods an extra round of messages is needed instead of writing to stable storage, so that enough replicas are informed of the effects of the transaction operations. It is generally agreed [4], [23] that stable-storage writes are approximately as costly as this additional messaging round. Hence, the execution of the 2PC protocol will introduce similar delay in both viewstamped and location-based replicated systems as in nonreplicated systems. The only drawback of the location-based method compared to other quorum-based methods is that a leader for a write operation will wait for a sufficiently long time at the prepare phase to allow all available replicas to respond. This additional delay, as we saw in Section VII, is more than offset by the asynchronous nature of operation execution and lock acquisition which saves significant communication cost for every transaction operation. However, recall that the location-based paradigm can be modified so that only A_w replicas, instead of all replicas, are involved in write operations. Thus, LBR will incur the same cost during 2PC as the PC protocol.

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

As shown in Section VI–A our paradigm provides smooth replication service, minimizing the disruption of service which occurs when failures (of sites and links) occur. Depending on when a failure occurs during transaction execution, the location-based paradigm introduces either no overhead, or, in the worst case, a two-round delay and one additional message. In the viewstamped method, for example, transactions executing in the midst of failures are interrupted by at least a three-round delay (two rounds for the view-change algorithm and one round for the message informing the primary of its role) and $2 \times n - 1$ additional messages, where n is the number of replicas. The location-based paradigm also employs an efficient recovery algorithm which facilitates independent crash-recovery and, thus, rapid re-introduction of a recovering site into the system. In contrast, in related work [8], [21], [23] either a one-phase (two rounds) or two-phase (four rounds) protocol must execute whenever a recovery occurs to re-introduce the recovering replica into the system.

No other replication method can claim performance similar to nonreplicated systems. Typically, in other methods [1], [8], [9], [12], [13], [18] reads and/or writes are applied to a number of replicas before the results are obtained. A notable exception is the method employed by the transactional version of ISIS [4], [16]. ISIS researchers proposed the concept of asynchronous operation execution. ISIS uses the notion of a replica coordinator. Operations are synchronously performed at the coordinator only and, like location-based replication and viewstamped replication, other replicas are notified asynchronously. However, ISIS requires an expensive, two-phase protocol for synchronous replicated write-lock acquisition and deadlock prevention. Thus, we expect our replication paradigm to exhibit better performance than the method used in ISIS. Furthermore, ISIS does not work well in the presence of partition failures.

In [29] we have shown that the basic principles of the location-based paradigm can be extended in order to further reduce the execution delays of transactions.

B. Availability Considerations

Recall from the interaction among transaction clients and the location service that only a single location-server replica needs to be available in order for a "server?" query to execute. In addition, "update-table" operations do not require any location-server replica to be available. We claim that the unavailability of location-server replicas does not unnecessarily limit the availability of data objects and operations. Let us illustrate our claim with a worst-case example. Suppose an object x, has five copies. Failures result in two partitions, P_1 with three copies and P_2 with two copies of x. Furthermore, P_1 contains only one location-server replica. Note that any table entry for x will have at least three sites associated with it since write operations are performed on at least a majority of replicas. A transaction client c_1 wishing to read x in P_1 will thus obtain a list of at least three replicas of x in response to its "server?" query. Thus, at least one of the replicas in the list must be in P_1 and P_1 contains at least one replica with the highest version number for x. Thus, c_1 will surely consult a replica that is in the list and in its partition. If this replica stores a stale version for x then during the prepare phase, the staleness will be detected, the transaction will abort, and c_1 will be notified of the replica that stores the most recent version. This will become the leader when the transaction is restarted and, as long as P_1 contains $A_r(x)$ replicas, the read operation will succeed.

While it is true that the client-location-server interaction may force some transactions to abort, in Sections IV and VII we saw that these abortions are very rare. Therefore, the availability of read (write) operations in the replication paradigm is principally constrained by the requirement that A_r (or A_w) replicas be available for each transaction to prepare and commit during 2PC.

Our method, like most quorum-based methods, is flexible enough to allow trade-offs between the availability of read and write operations. By manipulating the variables A_r and A_w for any object, we can trade-off the availability characteristics of read and write operations. This is not possible in the viewstamped replication method (which, as a result of the view-change algorithm requires a majority for reads and writes). Therefore, location-based replication is more flexible than viewstamped replication.

A few research efforts have been directed towards developing methods whose availability characteristics degrade gracefully in response to failures and, especially, network partitions. For example, El Abbadi and Toueg [7], [8] and Herlihy [13] recently presented methods in which quorum sizes are adjusted (as failures occur) so that the availability of operations remains high. 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 employ such methods.

The viewstamped method induces transaction abortions when nodes and communication links fail and a view change 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. Our paradigm does not depend on a view-change protocol and therefore reduces the disruption of service when failures occur.

IX. Conclusion

This paper contributes a new paradigm for replication. The paradigm, in essence, contradicts the 'conventional wisdom' which asserts that performance and availability are conflicting goals. In particular, it shows that the performance penalty for increasing availability can be kept small enough so that transaction response times can be similar to those in nonrepli-

cated distributed systems. In fact, we show that by facilitating localized execution of transaction operations it is possible to achieve better response times than would be possible in nonreplicated systems.

The location-based replication paradigm consists of three components. The first component is a location service with an extended functionality. Its novel aspects are the augmentation of the semantics of location information and its inexpensive incorporation into the transaction-processing mechanism. The end result is that transaction operations need only execute (synchronously) at a single replica site. The second component is an asynchronous concurrency-control algorithm. It contributes an efficient way of avoiding serializability problems while requiring only a single replica to synchronously acquire locks. The third component is a crash-recovery algorithm that exploits the benefits of the extended location service and allows independent recovery and rapid reintegration of the failed replicas.

Our paradigm avoids the inefficiencies associated with methods based on the primary-copy paradigm for replication. In particular, the latter may cause performance bottlenecks, disallows load balancing, and may exhibit considerable disruption of service (especially in environments where failures are more frequent). In addition, the primary-copy paradigm does not allow 'localized' execution of transaction operations. In distributed systems that span large geographical areas, communication with the primaries may introduce significant costs. Therefore, the primary-copy paradigm cannot satisfy one of the two motivations for introducing replication in a distributed system.

Our method has good availability characteristics, in that only $A_r(A_w)$ replicas are required to be available for read (write) operations. Thus, it has the same availability as quorum-based protocols. In addition, in [27] we have shown that the principles of the paradigm can be extended to achieve optimal availability efficiently.

Finally, we have conducted simulation studies to analyze the performance of the location-based paradigm for replication. We constructed, along with a performance model of this protocol, additional models representing well-known replication protocols and a model for nonreplicated systems. Our studies have shown that our protocol indeed enjoys shorter execution delays compared to the other replication control protocols and to nonreplicated systems.

REFERENCES

[1] D. Agrawal and A. El Abbadi, "The tree quorum protocol: An efficient approach for managing replicated data," in *Proc. Int. Conf. Very Large Data Bases*, Aug. 1990.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.

[3] P. Bernstein and N. Goodman, "The failure and recovery problem for replicated databases," in *Proc. 2nd ACM Symp. Princip. Distrib. Comput.*, Aug. 1983, pp. 114–122.

[4] K. Birman, "Replication and fault-tolerance in the ISIS system," in *Proc. 10th ACM Symp. Oper. Syst. Princip.*, Dec. 1985, pp. 79–86.

[5] A. Demers *et al.*, "Epidemic algorithms for replicated database maintenance," in *Proc. 6th ACM Symp. Princip. Distrib. Comput.*, Aug. 1987, pp. 1–12,

[6] D. Duchamp, "Analysis of transaction management performance," in *Proc. 12th ACM Symp. Operating Systems Principles*, Dec. 1989, pp. 177–190.

[7] A. El Abbadi and S. Toueg, "Availability in partitioned replicated databases," *ACM Trans. Database Syst.*, 14(2) pp. 264–290, June 1989.

[8] ——, "Availability in partitioned replicated batabases (extended abstract)," in *Proc. 5th ACM Symp. Princip. Database Syst.*, Mar. 1986, pp. 240–251.

[9] D. Gifford, "Weighted voting for replicated data," in *Proc. 7th ACM Symp. Oper. Syst. Princip.*, Dec. 1979, pp. 150–162.

[10] G. Gordon, *System Simulation*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[11] M. A. Greer, "Performance measurement of remote IPC", M.Math thesis, Dep. of Computer Science, Univ. of Waterloo (1989).

[12] M. Herlihy, "Concurrency versus availability: Atomicity mechanisms for replicated data," *ACM Trans. Comput. Syst.*, 5(3) pp. 249–274, Aug. 1987.

[13] M. Herlihy, "Dynamic quorum adjustment for partitioned data," *ACM Trans. Database Syst.*, 12(2) pp. 170–194, June 1987.

[14] ——, "A quorum consensus method for abstract data types," *ACM Trans. Comput. Syst.*, 4(1) pp. 32–53, Feb. 1986.

[15] ISIS, Modelling Communication Costs, discussion in the comp.sys.isis newsgroup, 1990.

[16] T. A. Joseph and K. Birman, "Low cost management of replicated data in fault-tolerant distributed systems," *ACM Trans. Comput. Syst.*, 4(1) pp. 54–70, Feb. 1986.

[17] W. H. Kohler, "A survey of techniques for synchronization and recovery in decentralized computing systems," *ACM Comput. Surveys*, 13(2), pp. 149–185, June 1981.

[18] A. Kumar, "Performance analysis of a hierarchical quorum consensus algorithm for replicated objects," in *Proc. 10th Int. Conf. Distrib. Comput. Syst.*, May 1990.

[19] R. Ladin, B. Liskov, and L. Shrira, "A technique for constructing highly available services," Laboratory for Computer Science, MIT, Tech. Rep. MIT/LCS/TR-409, Jan. 1988.

[20] B. Lampson, "Atomic transactions," in *Lecture notes in Computer Science vol. 105. Distributed Systems: Architecture and Implementation.* New York: Springer Verlag, pp. 246–265, 1981.

[21] D. D. E. Long, "The Management of Replication in a Distributed System," University of California, San Diego, Dep. of Computer Science, Ph.D. thesis (available as Tech. Rep. from the Univ. of California, Santa Cruz, UCSC-CRL-88-07) 1988.

[22] B. M. Oki and B. Liskov, "Viewstamped replication: A new primary copy method to support highly available distributed systems," in *Proc. 7th ACM Symp. Princip. Distrib. Comput.* Aug. 1988, pp. 8–17.

[23] B. M. Oki, "Viewstamped replication for highly available distributed systems," Lab. for Computer Science, Massachusetts Inst. of Technology, Ph.D. thesis, (also published as MIT/LCS/TR-423), Aug. 1988.

[24] C. Papadimitriou, "The serializability of concurrent updates," *J. ACM*, 26(4) pp. 631–653, Oct. 1979.

[25] J-F. Paris, "Voting with witnesses: A consistency scheme for replicated files," in *Proc. 6th Int. Conf. Distrib. Comput. Syst.*, May 1986, pp.606-612.

[26] D. J. Taylor, "How big can an atomic action be," in *Proc. 5th Symp. Reliability in Distrib. Software and Database Syst.*, Jan. 1986, pp. 121–124.

[27] P. Triantafillou and D. J. Taylor, "Efficiently maintaining availability in the presence of partitionings in distributed systems," in *Proc. 7th Int. Conf. Data Eng., IEEE*, Apr. 8–12, 1991, pp. 34–41.

[28] P. Triantafillou, "Employing replication to achieve efficiency and high availability in distributed systems," Ph.D. thesis, Univ.of Waterloo, July 1991, (available as Res. Rep. CS-91-28).

[29] P. Triantafillou and D. J. Taylor, "Multi-class replicated data management: Exploiting replication to improve efficiency," *IEEE Trans. Parallel and Distrib. Syst.*, pp. 121–138, Feb. 1994.

Peter Triantafillou (M'90) received the B.A. degree from York University, Toronto, Canada in 1986, the M.Sc. degree from the University of Western Ontario, London, Canada in 1988, and the Ph.D. degree from the University of Waterloo, Waterloo, Canada in 1991, all in computer science.

Since September 1991 he has been on the faculty of the School of Computing Science at Simon Fraser University where he is currently an Assistant Professor. He has also been appointed as an Assistant Professor in the department of Computer Engineering at the Technical University of Crete, in Greece. His research interests are in distributed systems. In particular, his research activities focus on highly-available distributed databases and file systems, on the construction of multi-systems (such as multidatabases), and on the construction of systems which support multiple notions of consistency for their applications.

Dr. Triantafillou is an affiliate member of the IEEE Computer Society and a member of the Association for Computing Machinery.

David J. Taylor (S'76–M'77) received the B.Sc. degree in mathematics from the University of Saskatchewan in 1972 and the M.Math. and Ph.D. degrees in computer science from the University of Waterloo in 1974 and 1977, respectively.

Since 1977 he has been a faculty member in the Computer Science Department at the University of Waterloo, where he is currently an Associate Professor. While on sabbatical leave, he has held visiting positions at the Computing Laboratory, University of Newcastle upon Tyne and at the Centre for Advanced Studies, IBM Canada, Toronto. His main research interests are in distributed computer systems and fault-tolerant computing. His current primary research focus is tools for debugging distributed applications; other major research activities have concerned data structures for fault-tolerant systems, replication in distributed systems, and interkernel protocols for distributed systems.

Dr. Taylor is a member of the IEEE Computer Society and the Association for Computing Machinery.