High Throughput Escrow Algorithms for Replicated Databases*
(Extended Abstract)

Narayanan Krishnakumar Arthur J. Bernstein
Dept. of Computer Science, SUNY at Stony Brook,
Stony Brook, NY 11794-4400.
email : {nkris,art}@sbcs.sunysb.edu
tel : (516) 632-8434

Abstract

The traditional correctness criterion in replicated databases is one-copy serializability. However, this criterion is sometimes restrictive and degrades performance. Recent research has therefore focused on utilizing application semantics to increase transaction throughput in certain high-performance applications. One such application involves resource allocation. To improve concurrency in such a system, the transaction escrow (TE) and site escrow (SE) algorithms have been proposed. In this paper, we present a generalized site escrow algorithm (GSE) that provides high site autonomy and throughput. GSE requires only a loose synchronization between sites, and employs the mechanisms of quorum locking and background gossip messages. We perform a comparison between GSE and TE, and outline regions in which GSE performs better. We also propose a family of hybrid algorithms that switch between GSE and TE under appropriate circumstances so that the benefits of both algorithms can be utilized. Finally, we present a variant of GSE that does not use locking.

Keywords : replication, resource allocation, replica control, escrow.

*This work was supported by NSF Grants No. CCR8701671 and No. CCR8901966

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

1 Introduction

Distributed databases can exhibit better performance than centralized databases since transactions can execute concurrently at different sites. However, data may then have to be accessed at a remote site, and this can result in message passing delays. Replicating the data reduces these delays: when a transaction wishes to read the data, an expensive remote access is not required. Replication also increases the availability of the data, since the data can be accessed even if a few sites fail. Thus, databases are often replicated for reasons of increased performance and availability. The correctness criterion commonly used by concurrency control techniques in replicated databases is the notion of *one-copy serializability* [BHG87] of transactions. In some applications, however, we can improve transaction throughput beyond what is realizable with the above approach. This paper deals with such applications and the techniques used to obtain high throughput in a replicated database.

The class of applications that we consider involves resource allocation. Assume that there is a single resource type, of which there are several indistinguishable instances. Let allocd denote the number of instances that have been allocated at any time, and let the total number of instances be denoted by tot. The statement of the problem is to ensure that the follow-

ing *allocation constraint* holds : allocd $\leq$ tot.

The above resource allocation problem can arise in several application domains. Such applications usually involve aggregate fields which are updated by positive or negative incremental changes (e.g. inventory control systems, airline reservation systems, banking applications). In an airline reservation system the seats are the resources and ideally, overbooking of these seats should not be possible. Another application is in relational database systems with integrity constraints. Suppose an integrity constraint states that there must exist at least one tuple in a given relation R satisfying a property P (existential quantification). If there are several tuples in R satisfying P, they can be considered available resource instances. The allocation constraint here is that the number of tuples satisfying P should be greater than or equal to one. (The ideas we present are easily extensible to the dual problem where there is a lower bound on the number of allocated resource instances.)

Transactions request and release resources using allocate and deallocate operations. A typical allocate operation is as follows, where n is the number of resources requested by a transaction.

```
PROCEDURE Allocate ( n : integer )
    IF ( allocd +n ≤ tot )
    THEN allocd := allocd + n
    ELSE print "Unable to allocate"
END Allocate
```

A deallocate operation decrements allocd unconditionally.

Suppose transactions frequently access allocd to allocate and deallocate resources (*i.e.* allocd is a 'hot-spot' data item). Consider two long-running transactions that each contain allocate operations, and assume that locking is used for concurrency control. Since allocd is read and modified by both transactions, one of the transactions may have to wait for the other to release its lock on allocd. A traditional locking algorithm would allow the release of locks only at commit time, which implies that the waiting transaction could be delayed for a long time. To increase throughput in such situations, the ideas of transaction escrow [ONe86] and site escrow [KS88, SS90] have been proposed. These methods make use of the semantics of the application, and allow transactions to co-ordinate access to hot-spot data items without having to wait till other transactions commit. In this paper, we extend these techniques. The algorithms that we present use the mechanisms of background gossip messages and quorum consensus to co-ordinate transaction execution.

2 Related work

Consider an airline reservation system with 9 sites and a single flight which can carry at most 200 passengers. Let resrvd_seats be a data item denoting the number of seats reserved on the flight and assume that resrvd_seats is replicated at all sites. The allocation constraint is given by {resrvd_seats $\leq$ 200}.

Let us review the approaches that have been proposed to increase concurrency in such a situation :

1. The Transaction Escrow Algorithm [ONe86]: The transaction escrow algorithm (TE) has been proposed for access to hot-spots in a single copy database. In this algorithm, a transaction executes an *escrow* operation to try to place in reserve the resources that it will (potentially) use. All successful escrow operations are logged. Before executing an escrow operation, each transaction accesses the log and sees the total escrow quantities of all uncommitted transactions. The transaction then makes a worst-case decision to determine whether it can proceed. For instance, in the airline example, assume that the number of seats reserved due to committed transactions is 20. Suppose there are two uncommitted transactions each reserving one seat. Let transaction T wishing to reserve ten seats now be initiated. Since the log indicates that resrvd_seats is at most 22, T can proceed irrespective of whether the other two transactions commit or abort: the allocation constraint is maintained in any case.

Note that when transaction T executes an escrow operation, T (short-term) locks the data item to access/update the log and releases the lock after the log has been updated (the lock release need not wait for the commit or abort of T as in a traditional transaction execution). This feature allows long-running transactions that contain escrow operations to run concurrently so that throughput is increased.

A solution for the replicated case is not proposed in [ONe86]. Let us extrapolate the method to allow for replication. We use quorum locking [Her87] for the purpose of giving a decentralized solution. Before executing an escrow operation from transaction T, site i (short-term) locks a quorum of sites. Members of

the quorum send their logs to i. T then uses the allocations of previously committed transactions and the escrow quantities of all uncommitted transactions to determine whether it can proceed. If it can, i records the escrow operation in the log, sends this log to the quorum sites and unlocks the quorum sites (the locks are not held until commit/abort time). Observe that T has to see the requests of all committed and uncommitted transactions that "precede" it, so the quorums of these transactions must intersect with T's quorum. Since all the quorums are of the same size, each site has to lock a majority of sites. This lack of site autonomy can result in delays in processing transactions.

2. Site Escrow Algorithms : In TE, each *transaction* places in escrow the resources that it is attempting to acquire. In the site escrow algorithms (SE), each *site* places resources in escrow. A transaction can succesfully complete at a site only if the number of resources escrowed at that site exceeds the number of resources that the transaction requires. The escrowed quantity is then adjusted to reflect the consumption of resources by the transaction. This approach results in more site autonomy than TE, since each site can deplete its escrowed resources independently without having to consult any other site.

[KS88, Har88, SS90, BG91] present SE algorithms for maintaining the following classes of constraints, where B is a replicated numeric object, and B_{min} and B_{max} are constants: (a) $B \geq B_{min}$, and (b) $B \leq B_{max}$. We discuss the solution given in [KS88] to preserve the second constraint. The global state of the system is computed periodically by executing a global snapshot algorithm. Let M be the total number of sites in the network, and let B_{glob} indicate the value of B in the last recorded global state. The escrow quantity, es_i, allotted to each site i immediately after i has been informed of the global state is :

$$es_i = \frac{B_{max} - B_{glob}}{M}$$

If i cannot execute a transaction on the basis of es_i, i requests other sites to donate a portion of their escrow quantities for its use.

Since transactions cannot be processed while (one phase of) the global state algorithm is being executed, it is desirable not to execute it frequently. On the other hand, it is also desirable to have the algorithm executed frequently so that sites have a fairly accurate estimate of the global state:

1. Since es_i varies dynamically, the algorithm is adaptive : even if one site is doing most of the allocations, it can do so with a high degree of autonomy if frequent global snapshots are taken.
2. If there are decrements to B, the escrow quantities at all sites can be increased, so that site autonomy can be increased. However, the increase in the escrowed quantity can be done only when the global state is recomputed. The sooner the algorithm is executed, the sooner the increase in site autonomy, and hence better performance.

Thus for the purpose of performance, the global state algorithm has to be executed as frequently as possible, whereas the frequent execution of the algorithm may itself degrade performance. We attempt to address such shortcomings in this paper.

In [KB91], we introduced the notion of bounded ignorance, which allows a transaction to be ignorant of the effects of a limited number of transactions that precede it in the serial order. This increases concurrency at the cost of a bounded violation of the integrity constraints of the system. For instance, in a relational database with a student-advisor relation, the constraint that every student has an advisor (universal quantification) can be relaxed to the constraint that at most k students do not have an advisor. We can treat this as a resource allocation problem in which k is the total number of resource instances. By using escrow-based ideas, we can dynamically change the level of concurrency of updates to the student-advisor relation and this can result in higher throughput. We thus find that a traditional application can sometimes be cast into a resource allocation-style problem, and our techniques help improve site autonomy and concurrency in such applications.

3 The Model

We assume that the system consists of a set of sites that communicate by sending messages over a communication network. For simplicity, we assume that there are no site/communication failures (our algorithms, however, function correctly even in the presence of such failures). Suppose there is a single resource type, r. The *allocation constraint* is given by allocd $\leq$ tot, where tot is the total number of indistinguishable instances of r and allocd is a data item

denoting the number of instances of r allocated globally at any time.

In order that sites can make autonomous decisions concerning resource allocation, allocd is replicated at all sites. We refer to the value of a particular replica at a site as the *site view* at that site. A site view need not necessarily be up-to-date: it may be missing the updates of (a limited number of) transactions which have executed at other sites. Since we are interested in how the allocation constraint is maintained, we concentrate on the allocate and deallocate operations that are embedded in a larger transaction and loosely refer to these operations individually as *allocate* and *deallocate* transactions. (The results can easily be extended to the general case where there are several resource types, and transactions contain several steps that allocate and/or deallocate resources or access non-resource type data items.)

Suppose transaction T is executed at site i. We denote the number of resources which can be allocated (deallocated) by T as $req(T)$. We say that T has been *submitted* at i when T first begins execution at i. Execution proceeds in the following steps:

1. A communication phase with some sites may be required to ensure that i's site view is ignorant of only a limited number of allocations that have executed at other sites.
2. The read phase of T, RP_T, is executed during which i's site view is read. RP_T generates an unconditional update u_T that preserves the constraint if executed on the site view. We say that T is *initiated* when RP_T starts execution.
3. u_T is installed atomically in the local log for resource r.
4. A communication phase with some sites might be required to ensure that u_T is also installed in the log for r at those sites. u_T is also asynchronously broadcast to all other sites.

Observe that the update u_T is 'externalized' to other transactions, as in TE, before the larger transaction in which T is embedded commits. (If the larger transaction aborts, u_T has to be purged from the sites at which it has been logged. We do not discuss any details of how this takes place.)

For convenience, we assume that the state of the database is represented by a history - the logged sequence of updates. (It is sufficient to use a compacted version of the entire history, but the details are beyond the scope of this paper.) If T' and T are transactions, we say that $RP_{T'}$ *sees* the update u_T (or equivalently T itself) if u_T is in the history corresponding to the site view read by $RP_{T'}$. We define a happened-before partial ordering, $\rightarrow$, between transactions: $T \rightarrow T'$ if $RP_{T'}$ sees u_T. (In general, transactions can consist of updates to several resource-type data items. For each such data item, there is an independent happened-before ordering over the updates that modify that data item.) A transaction T is said to be *concurrent* with T', denoted T conc T', iff $T \not\rightarrow T'$ and $T' \not\rightarrow T$. We assume that the broadcast mechanism for disseminating updates satisfies the property that sites learn of updates which have been generated at other sites in accordance with $\rightarrow$. Thus, if a site knows of an update u_T it also knows of all updates generated by transactions T' such that $T' \rightarrow T$.

The transactions are ordered by a linear order $<_l$, which is a linearization of $\rightarrow$. $<_l$ is constructed using timestamps assigned to the transactions. If $TS(T)$ denotes the timestamp of transaction T, then $T_1 \rightarrow T_2$ implies $TS(T_1) < TS(T_2)$, and $T_1 <_l T_2$ if and only if $TS(T_1) < TS(T_2)$. Although updates might arrive at different sites in different orders (but satisfying $\rightarrow$), it follows that if the same set of updates arrives at two sites, they will yield the same site view since increment/decrement updates commute.

The *global state* of the database at any time t is the state that includes all updates that have occurred at all sites until time t. We say that the system is *correct* if the allocation constraint is true of the global state at all times. To ensure that the system does not behave 'incorrectly', a *replica/concurrency control algorithm* is required to co-ordinate concurrent access to the replicas by several transactions. In our case, the generalized site escrow algorithm described in Section 4 places a bound on how out of date the site view is allowed to be when a transaction is initiated. Step 1 above delays the read phase of the transaction when this bound is exceeded.

4 The Generalized Site Escrow algorithm

The Generalized Site Escrow (GSE) algorithm modifies the notion of site escrow in [KS88], and eliminates

the need for a global state algorithm to replenish the escrow quantity. Consider a particular scenario to motivate the approach. Assume that only a single site, i, is executing allocate transactions, and that i periodically broadcasts the allocations it has made to other sites. Suppose the escrow quantity at each site is computed using $es_i = \frac{Avail_i}{M}$, where $Avail_i$ is the number of resources that are recorded as available in i's site view. i then knows that the escrow quantity at each other site is dynamically being reduced as more allocate transactions are executed at i. If i can place a lower bound on what other sites know about allocations it has made, it can place an upper bound on their current escrow quantities and thereby dynamically replenish its own escrow quantity. i can then continue to allocate resources without stopping to execute the global state algorithm. Thus, to initiate a transaction using GSE, it is sufficient for i to know that other sites know of all but a certain number of past allocations seen at i. The synchronization required here is weak compared to that required by SE and this can result in high throughput.

The escrow quantity allotted to each site determines the site's autonomy and hence the level of synchronization between sites. To implement this synchronization GSE employs two mechanisms : (a) quorum locking [Her87] to control the number of allocations that can simultaneously be performed across the network and (b) broadcast using gossip messages to limit the extent to which a site may be ignorant of the effects of prior transactions done elsewhere.

In TE the quorums of two transactions have to intersect, and thus a quorum has to contain at least a majority of sites. In GSE quorums need not necessarily intersect, and hence can contain less than a majority of sites. This allows multiple allocations/deallocations to be occurring simultaneously in the net.

Gossip messages are background messages which can be used to propagate information regarding what a site knows to other sites in the network. Gossip messages ensure that a site learns of updates in the $\rightarrow$ order. Several algorithms have been proposed which allow a site to maintain information about the state of another site's knowledge ([WB84, HHW89]). The technique involving the use of a *timetable* is described in [WB84]. The timetable at site i, TT_i, can be characterized as follows:

1. $TT_i[i,i]$ is a counter incremented whenever a transaction T is initiated at site i. Denote the kth row of TT_i as $TT_i[k]$. $TT_i[i]$ is assigned as the vector timestamp, $TS(T)$, of T.
2. If $TT_i[j,k] = x$, then site i knows that site j knows of all transactions initiated at site k when $TT_k[k,k]$ was less than or equal to x.

---

0. $Q_T = \{i\}$
1. (Receive gossip messages)
2. $es_i = \lfloor \frac{Avail_i}{M} * |Q_T| \rfloor$
3. Compute $\mathcal{U}_i$
4. IF $es_i < req(T) + req(\mathcal{U}_i)$
   THEN IF ($|Q_T| < M$)
   THEN
   Lock a site j and add to Q_T.
   Goto Step 1
5. Increment $TT_i[i,i]$ and create a timestamp for T.
6. Execute RP_T on i's view and append u_T to view.
7. Unlock quorum sites.

Figure 1: Algorithm GSE executed by i to initiate T

Each gossip message sent by site j to site i at time t contains a snapshot, ST_j, of TT_j, and all the updates that j has seen up to time t. We can optimize the amount of information sent in a message [WB84] by (1) using the timetable to reason about the knowledge state of the destination site (*i.e.* if j knows that i knows of some updates, then it need not include those updates in its gossip message to i), and (2) sending only a portion of the timetable. In a system with a large number of sites and in which a large number of updates occur, only the optimized approach is feasible. We do not however discuss such optimizations here.

Assume that a gossip message m is sent from site j to site i. Site i merges the update information in m into its local history on the basis of the timestamps. TT_i is updated in the following fashion:

M1. $\forall p \in \mathcal{SI}$ do $TT_i[i,p] := max(TT_i[i,p], ST_j[j,p])$. This indicates that for each site p, site i now knows the updates of all the transactions initiated at p that site j knew about (when the gossip message was sent).

M2. $(\forall p \in \mathcal{SI})$ $(\forall n \in \mathcal{SI})$ do $TT_i[p,n] := max(TT_i[p,n], ST_j[p,n])$. This indicates that the transactions that j knew were initiated at n and

have been seen at p (when j sent the gossip message), are also known by i to have been seen at p. Thus, i is brought up-to-date concerning the knowledge states at the other sites.

Lemma 1 *Suppose site i knows at (global) time t that site j knows of some update u_T. Then there exists a chain of gossip messages from j to i such the first message in the chain was sent from j after it knew of u_T and the last message is received at i before t.*

Proof of Lemma 1: See [KB91]. ■

The following lemma lets a site use its timetable to reason about what it knows of the state at other sites. Let a and b be two vectors. Define $a \preceq b$ if and only if a is elementwise less than or equal to b.

Lemma 2 *Consider a transaction T and a site i. $TS(T) \preceq TT_i[k]$ if and only if i knows that u_T is in the site view of k.*

Proof of Lemma 2: See [KB91]. ■

The mechanisms of quorum locking and gossip messages are integrated into GSE in the following fashion. The messages that are propagated in the system as part of the quorum algorithm are : (a) Lock request, (b) Lock grant, (c) Lock release, and (d) Lock deny. Gossip messages are piggy-backed onto these messages. We assume that gossip messages are also transmitted periodically by a site to neighboring sites independent of the quorum messages.

GSE is presented in Figure 1. The total number of sites in the system is denoted by M. es_i is the escrow quantity at site i, and $Avail_i$ is the number of available resources corresponding to i's view of $tot-allocd$. The values of $Avail_i$ and es_i are recomputed at the beginning of each iteration of the loop in Steps 1 through 4. We assume that gossip messages can be received only at Step 1, but there is no restriction on when gossip messages can be sent from i (the restriction on the receives is for clarity only: Steps 2 and 3 are steps that compute values based on the site view at Step 1. i can in fact receive gossip messages while trying to lock j in Step 4). For simplicity, we assume that each site includes itself in its quorum so that a site can only initiate one transaction at a time.

Suppose i wishes to execute transaction T. We denote T's quorum by Q_T (the size of the quorum is $|Q_T|$). Initially, only i is in Q_T. If all sites were completely up-to-date then, for all i and j, $Avail_i = Avail_j$ and each would take $\lfloor \frac{Avail_i}{M} \rfloor$ as its escrow quantity. Site i can utilize the escrow quantities of sites in Q_T since they are locked and hence es_i is given by $\lfloor \frac{Avail_i}{M} * |Q_T| \rfloor$.

Unfortunately, sites are not up-to-date concerning transactions that have executed at other sites and hence i must make a conservative estimate of the escrow available to it to ensure that concurrent allocations do not cause a violation of the allocation constraint. It does this by using an upper bound on the possible escrow values seen by non-quorum sites. There are two factors that can cause the escrow seen at non-quorum sites to be larger than es_i : (1) allocate transactions known to i that are unknown at non-quorum sites and (2) deallocate transactions known at non-quorum sites but unknown at i. With respect to factor (2), a transaction deallocating s resource instances at non-quorum site j only enlarges es_j by $\lfloor \frac{s}{M} \rfloor$ (j's portion of the deallocated quantity). Hence any resulting additional allocation performed at j concurrent to T cannot cause a violation of the constraint.

We now deal with factor (1). Let $\mathcal{U}_i$ denote the set of all allocate transactions, T', such that T' is known to i and i is not certain that T' is known to all the non-quorum sites *i.e.* from Lemma 2, there exists a non-quorum site k such that $(TS(T') \not\preceq TT_i[k])$. For a set of transactions $\mathcal{S}$, let $req(\mathcal{S})$ denote $\sum_{T' \in \mathcal{S}} req(T')$. Note that if T is executed, the allocation of as many as $req(\mathcal{U}_i) + req(T)$ resources might be unknown to non-quorum sites. As a result, the non-quorum sites might have overestimated their respective escrow quantities and executed transactions concurrent with T on the basis of those escrows. By ensuring that $req(\mathcal{U}_i) + req(T)$ is not larger than $\lfloor \frac{Avail_i}{M} * |Q_T| \rfloor$ (which is i's available escrow quantity, es_i), we can guarantee that the allocation constraint is preserved (as proved in Section 6). Thus, before T can be initiated, i confirms in Step 4 that all but at most the last $es_i - req(T)$ allocations seen at i are known to all non-quorum sites (i.e. $req(\mathcal{U}_i) \leq es_i - req(T)$). If the inequality in Step 4 is true, then i's escrow is insufficient and i has to enlarge Q_T by locking additional sites. es_i and $\mathcal{U}_i$ are then recalculated, and the test is again made to see if the escrow quantity is enough. In the worst case, all M sites are in Q_T, at which point T can be initiated. (We propose some optimizations later in the paper to decrease the size of Q_T.)

For simplicity, our description of GSE assumes that i locks quorum sites in a sequential manner.

i can, however, accumulate the quorum in parallel. i estimates the required quorum size as $|Q_T| = \left\lceil \frac{(req(T)+req(\mathcal{U}_i))*M}{Avail_i - req(\mathcal{U}_i)} \right\rceil$ and locks $|Q_T|-1$ (other) sites in parallel to form the quorum. If the resulting quorum is insufficient it is enlarged by locking additional sites.

Note that site i is tightly synchronized with other sites in Q_T, but only loosely synchronized with non-quorum sites.

5 Discussion

The benefits of GSE are as follows :

1. GSE requires no global synchronization points at which the escrows are recalculated, hence an algorithm that explicitly computes the global state is not required. Using background messages, GSE assures a lower bound on what a site knows about transactions executed at other sites.
2. A quorum is frequently smaller than a majority of sites (unlike in TE). The exact quorum size depends on $Avail_i$, and its minimum value occurs when $\mathcal{U}_i$ is null. In that case, GSE requires that es_i simply be greater than or equal to $req(T)$. If $Avail_i \geq req(T) * M$, the minimum quorum size is one. If $\frac{req(T)*M}{q-1} > Avail_i \geq \frac{req(T)*M}{q}$, where q is an integer larger than 1, the minimum quorum size is q. Consider the airline example where at most 200 seats can be reserved and $M = 9$. A transaction at site i wishing to reserve a single seat need only include i in its quorum if $Avail_i$ is less than 191 and $\mathcal{U}_i$ is null. If a transaction wishes to reserve 3 seats when $Avail_i$ is 183, then only 2 sites need to be in its quorum. In most cases where the number of allocations being requested by a transaction is small and there are several available resource instances, only the site initiating the transaction is in the quorum. Thus we expect the average execution time of transactions to be small.
3. The larger the value of $req(\mathcal{U}_i)$, the larger the value of es_i required and hence larger the delay before T can be initiated. By increasing the frequency with which gossip messages are exchanged, these allocations become known to other sites more quickly, thereby reducing the required value of es_i. As a result, the quorum size and consequently the response time for T is reduced.
4. GSE adapts to a situation in which a single site, i, does most of the allocations. Here, $Avail_i$ does not change significantly as a result of information about new allocations at other sites, and $req(\mathcal{U}_i)$ is small if gossip messages are frequent enough. i is then able to execute transactions with a high degree of autonomy. SE [KS88] also adapts to such an unbalanced allocation distribution. In fact, GSE reduces to SE in the special case that gossip message exchange is restricted to periodic intervals during which transaction execution is halted and in that interval the exchange of gossip messages brings all sites up-to-date.

In our description of GSE, deallocate transactions are not included in $\mathcal{U}_i$, and deallocations at i can be done without consulting any site. If, however, there is a lower bound, *low*, on the number of allocated resources, deallocate transactions can be synchronized independently through a dual of GSE: es_i is evaluated using the formula $\lfloor \frac{tot-low-Avail_i}{M} * |Q_T| \rfloor$, and $\mathcal{U}_i$ is the set of deallocate transactions that i believes is not known to non-quorum sites.

6 Correctness Proof

In this section, we show that GSE preserves the allocation constraint at all times. We say that a transaction, T_m, is a *maximal element* of $\rightarrow$ if there exists no transaction T' such that $T_m \rightarrow T'$. Consider the set, $\mathcal{M}$, of maximal elements of $\rightarrow$. By definition, any two elements in $\mathcal{M}$ are concurrent to one another and their quorums are disjoint: if T and T' are distinct elements in $\mathcal{M}$ such that $Q_T \cap Q_{T'} \neq \phi$, then either $T \rightarrow T'$ or $T' \rightarrow T$, which is a contradiction. Let T_i, $1 \leq i \leq n$, be the elements of $\mathcal{M}$, and assume that T_i is initiated at site i (at most one element of $\mathcal{M}$ can be initiated at site i). For simplicity, we assume that all transactions in the system are allocate transactions (the results do not change if deallocate transactions are also considered).

Let $\mathcal{S}_i$ be the set of transactions seen by T_i (at Step 5 of GSE). Then the set of all transactions executed in the system is $\bigcup_{j=1}^{n} \mathcal{S}_j \cup \bigcup_{j=1}^{n} \{T_j\}$. Thus, the allocation constraint for the system can be written as :

$$tot \geq req(\bigcup_{j=1}^{n} \mathcal{S}_j) + req(\bigcup_{j=1}^{n} \{T_j\}) \qquad (6.1)$$

Let SS_i be the set of all transactions T' such that T' is in $\mathcal{S}_i$ and i knows before Step 5 of the algorithm that all the non-quorum sites know about T'. Thus $req(\mathcal{S}_i) = req(\mathcal{SS}_i) + req(\mathcal{U}_i)$. At most $es_i - req(T_i)$ allocations from $\mathcal{S}_i$ are not known to the non-quorum sites when T_i is initiated: $req(\mathcal{U}_i) \leq es_i - req(T_i)$. Thus $req(\mathcal{S}_i) \leq req(\mathcal{SS}_i) + es_i - req(T_i)$. Denote by RS the quantity $req(\bigcup_{j=1}^{n} \mathcal{SS}_j)$. From (6.1), the allocation constraint is satisfied if

$$tot \geq req(\bigcup_{j=1}^{n} \mathcal{SS}_j) + \sum_{j=1}^{n}(es_j - req(T_j)) + \sum_{j=1}^{n} req(T_j)$$

$$i.e.\ \ tot \geq RS + \sum_{j=1}^{n} es_j \tag{6.2}$$

Note that a particular transaction might be included in both $\mathcal{U}_i$ and $\mathcal{U}_j$, $i \neq j$, and therefore be included in both es_i and es_j. Thus the transaction may be counted twice in the above summation. However, it is sufficient that (6.2) be maintained for the allocation constraint to hold.

Observe that for each $j \neq i$, $1 \leq j \leq n$, all the transactions in the set $\mathcal{SS}_j$ are known to i before T_i is initiated. To see this, assume the contrary: there exists a transaction, T', such that i knows of T' only after u_{T_i} has been logged. By definition, j knows that $\mathcal{SS}_j$ is known to i before it initiates T_j. Then from Lemma 1 and our assumption, T_i is known to j before T_j is initiated, and $T_i \rightarrow T_j$ which is contrary to assumption that both T_i and T_j are maximal elements in $\rightarrow$. Thus for all j, $1 \leq j \leq n$, $\mathcal{SS}_j \subseteq \mathcal{S}_i$, so that $\bigcup_{j=1}^{n} \mathcal{SS}_j \subseteq \mathcal{S}_i$. Therefore, $req(\mathcal{S}_i) \geq req(\bigcup_{j=1}^{n} \mathcal{SS}_j) = RS$. Since before initiating T_i, $Avail_i = tot - req(\mathcal{S}_i)$, we have $Avail_i \leq tot - RS$. Hence, at Step 5, es_i satisfies

$$es_i \leq \frac{(tot - RS)}{M} * |Q_{T_i}|.$$

and the algorithm guarantees that for all i, $1 \leq i \leq n$,

$$tot \geq es_i * \frac{M}{|Q_{T_i}|} + RS \tag{6.3}$$

By comparing (6.2) and (6.3), it can be seen that (6.2) is true if for some i,

$$es_i * \frac{M}{|Q_{T_i}|} \geq \sum_{j=1}^{n} es_j,$$

or

$$es_i * (\frac{M}{|Q_{T_i}|} - 1) \geq \sum_{j \neq i} es_j \tag{6.4}$$

Without loss of generality, let site 1 be such that for all sites j $(2 \leq j \leq n)$,

$$es_1 * \frac{|Q_{T_j}|}{|Q_{T_1}|} \geq es_j$$

This implies that

$$es_1 * (\frac{\sum_{j=2}^{n} |Q_{T_j}|}{|Q_{T_1}|}) \geq \sum_{j=2}^{n} es_j$$

Since the quorums of the elements of $\mathcal{M}$ are mutually disjoint, we have that $\sum_{j=2}^{n} |Q_{T_j}| \leq M - |Q_{T_1}|$, which yields the required result (6.4).

Theorem 1 *Algorithm* GSE *preserves the allocation constraint.* ■

7 Comparison of GSE and TE

We can compare GSE and TE approximately by evaluating the times required to execute a transaction T (from submission of T to when T's site of initiation knows that sites in Q_T have been unlocked).

We assume that each message is the same length. Let maj denote $\lfloor \frac{M}{2} \rfloor + 1$, and $C(n)$ the (communication) time required by any site to send messages to or receive messages from n other sites. We assume that C is a linear function of n. (This is a worst-case assumption that is true when the network has a linear topology.) We assume that the time required to execute RP_T and append u_T to the log is negligible compared to the communication time.

Assume that T executes at site i. In TE, a majority quorum is locked before T is initiated. The minimum time required to execute T is the time for locking and unlocking the quorum sites and is given by $4 * C(maj - 1)$. In GSE, site i makes an estimate of the required quorum size based on its current site view, and then locks that many sites in parallel. We assume that the estimate is accurate with high probability. Thus, in GSE, the time required to execute T is $4 * C(|Q_T| - 1)$. (In both cases, we have not taken into account the time that i might spend in waiting for a lock at a possible quorum site to be released. We assume that

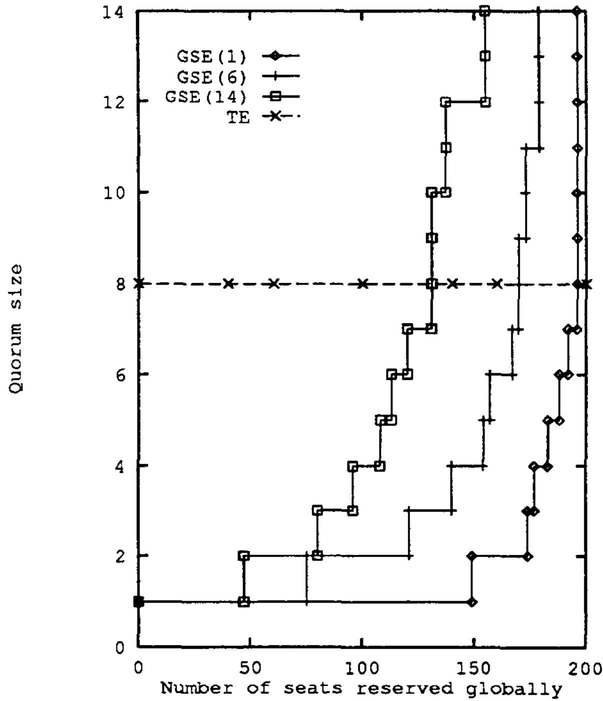

Figure 2: Simulation of GSE on 14-site network - GSE(n) indicates n sites can generate transactions simultaneously

this time is the same in both cases on the average.) If the quorum size is less than a majority, GSE performs better than TE. However, GSE requires that at least a majority of sites be included in the quorum when $Avail_i \leq 2 * (req(T) + req(\mathcal{U}_i))$. In this region, GSE performs worse than TE. To improve the efficiency of GSE in such cases, we propose some optimizations in the next section to reduce $|Q_T|$.

When $Avail_i \geq (req(T) + req(\mathcal{U}_i)) * M$, $Q_T = \{i\}$. This case occurs when the total number of available resources is large compared to both $req(T)$ and M, and the gossip messages are propagated frequently. Notice that in this important case, the time required to execute T is only the time for executing RP_T and appending u_T to the log. Consider an instance of the airline example where resrvd_seats$_i$ is 50 and T wishes to reserve 5 seats. Let i know about 11 seat allocations that it is not sure all other sites know about *i.e.* $req(\mathcal{U}_i) = 11$. Then $req(T) + req(\mathcal{U}_i) = 16$, and the quorum size required is 1 ($Q_T = \{i\}$).

Figure 2 shows the results of a simulation on a 14 site network (in which each site has at most degree 4). We plot $|Q_T|$ as a function of the number of seats reserved globally (resrvd_seats), with the number, n, of sites simultaneously generating transactions as a parameter. Transactions are generated at equal frequency at these sites, and gossip messages are transmitted from all sites at twice this frequency. The number of seats reserved in each transaction is a random number between 1 and 5. It can be seen from the plots that if $n = 1$ (only one site is initiating transactions), $|Q_T|$ remains 1 until resrvd_seats becomes ~150. If $n = 14$, $|Q_T|$ remains 1 until resrvd_seats becomes ~50 and remains less than maj (= 8 sites) until resrvd_seats becomes ~130.

8 Optimizations

8.1 Using information about other sites to reduce quorum size

The size of the quorum required to execute a transaction can be reduced by making use of the timetable, which bounds the knowledge state of other sites.

Assume that transaction T has been submitted at site i and that i has locked at least maj sites, but the quorum is still not sufficient. Let $Avail_i$ indicate the number of available resources seen at i at this stage. For simplicity, assume that T is not concurrent to any deallocate transaction (the result holds even if this assumption is not true). Using its timetable, i can place a lower bound on which transactions are known at each non-quorum site, and hence place an upper bound on $Avail_j$ for each non-quorum site j as follows. Denote by MA the maximum of these upper bounds. Let D_i indicate the number of deallocations that i has seen but which i is not sure that all the non-quorum sites have seen. Then, for any non-quorum site j, $MA + D_i$ is an upper bound on $Avail_j$. Since $M - |Q_T|$ is the number of non-quorum sites, $\lfloor \frac{MA+D_i}{M} * (M - |Q_T|) \rfloor$ is an upper bound on the sum of the escrow quantities at the non-quorum sites, and hence indicates the maximum number of allocations that can occur at non-quorum sites concurrent to T. If the number of available resources seen at i is at least the sum of the escrow quantities at non-quorum sites, the allocation constraint is preserved. Thus, if

$$Avail_i - req(T) \geq \left\lfloor \frac{MA + D_i}{M} * (M - |Q_T|) \right\rfloor \quad (8.1)$$

is satisfied, i can initiate T. (The idea depends on

the fact that at most one site can be doing the optimization, since at most one site can have locked *maj* sites.)

Example 1 : To illustrate the result, consider the following scenario of the airline reservation system for the case $MA = Avail_i$ (similar scenarios can be given for $MA < Avail_i$ and $MA > Avail_i$). Assume that i knows that the site view (resrvd_seats) at every other site is at least 199. If transaction T at site i wishes to reserve the last seat, then using GSE, all 9 sites in the system have to be in Q_T. Using (8.1), however, i needs to lock only 5 (=*maj*) sites. The intuition behind this situation is that any transaction that tries to reserve the last seat must lock at least a majority of sites, and locking a majority is sufficient for correctness.

Theorem 2 *Suppose site i wishes to initiate transaction T and has locked at least maj sites. If (8.1) is true, i can initiate T and the allocation constraint is still preserved.* ∎

8.2 Switching between GSE and TE

The optimization in the previous section is based on (8.1) being satisfied and thus cannot guarantee that a quorum consists of at most *maj* sites. However, by placing an additional restriction on the system, we can design hybrid algorithms in which a site switches from GSE to TE and back depending on its site view. We can then guarantee that a site needs to be synchronized with no more than *maj* sites before it can initiate a transaction.

Assume two positive integers b_1 and b_2 such that for any T, $req(T) \leq b_1$ and the initiation of T is delayed if $req(\mathcal{U}_i)$ exceeds b_2. (The extent of the delay can be reduced by increasing the frequency of gossip messages.) Let $b = b_1 + b_2$. Notice that es_i needs to be only as large as b for T to be initiated. Under these assumptions, a site i using GSE requires a quorum of at least *maj* sites only when $Avail_i < 2 * b$. To avoid using more than *maj* sites, consider an algorithm in which i switches to TE at this point. Note that another site j might still be using GSE since $Avail_i$ and $Avail_j$ need not be equal.

In this context, given a set of algorithms, we can associate an *efficiency predicate* with each to indicate which algorithm(s) is best to use for a particular site view. For instance, we can associate the efficiency predicate $\{ Avail_i \geq 2 * b \}$ with GSE and $\{ Avail_i < 2 * b \}$ with TE. Denote an algorithm A with an efficiency predicate P by the tuple $\langle A, P \rangle$. Assume that a site uses A if P is true of its site view. Given $\langle A1, P1 \rangle$ and $\langle A2, P2 \rangle$, each site can use one of A1 or A2 depending on whether P1 or P2 is true of its view (if both P1 and P2 are true, either algorithm can be used). Since site views can differ, some sites could be using A1 and others A2 at the same time. We say that $\langle A1, P1 \rangle$ is *compatible* with $\langle A2, P2 \rangle$ if some sites can be using A1 at the same time as others are using A2, and the allocation constraint is still preserved.

If $\langle A1, P1 \rangle$ and $\langle A2, P2 \rangle$ are incompatible, switching cannot occur since the allocation constraint can be violated. Example 2 shows that $\langle GSE, \{ Avail_i \geq 2*b \} \rangle$ is incompatible with $\langle TE, \{ Avail_i < 2 * b \} \rangle$:

Example 2 : Consider the following scenario of the airline example. Assume that $b = 5$, so that a site i switches to TE when it detects that $Avail_i < 10$. Let the global state be 191 (seats reserved) and let all site views see this state. Subsequently, let 5 seat cancellations occur and assume all become included in the site views of sites 6 through 9. Thus $Avail_i = 9$ for $1 \leq i \leq 5$ while $Avail_i = 14$ for $6 \leq i \leq 9$. Now assume T is submitted at site 1 such that $req(T) = 5$. Then using TE, site 1 can successfully reserve 5 seats by locking sites 1 through 5 (a majority). Site 1 can then execute another transaction reserving 4 seats using TE while locking sites 1 through 5. Thus $Avail_i = 0$ for $1 \leq i \leq 5$. If the allocation constraint is to be maintained, we have to show that the sites 6 to 9 cannot reserve more than 5 seats, which is exactly the number of cancellations that have been seen by them. Assume now that sites 6 and 8 each want to reserve 3 seats. They will use GSE with a quorum of size 2 since $Avail_6 = Avail_8 = 14$. Site 6 can include site 7 in its quorum, and likewise site 8 can include site 9 in its quorum. The resultant state has 201 reserved seats which is a violation of the allocation constraint. ∎

In fact, it can be shown [KB92] that there exists no positive integer h such that $\langle GSE, \{ Avail_i \geq h \} \rangle$ is compatible with $\langle TE, \{ Avail_i < h \} \rangle$.

Consider the algorithm, MAJ, which is exactly GSE except that Q_T always has exactly *maj* sites, and if the condition in Step 4 of GSE is true, then either i waits for the condition to become false (it does not include more sites in the quorum) or aborts the transaction. Note that MAJ is less efficient than GSE since it locks more sites than GSE would. Furthermore, in contrast

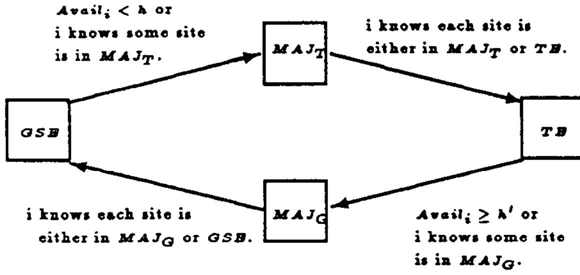

Figure 3: State transition diagram for HE at site i

to TE, MAJ cannot allow a transaction to be initiated if it has locked maj sites but the condition in Step 4 is true.

For all predicates P1 and P2, it can be shown that $\langle$MAJ, P1$\rangle$ is compatible with $\langle$GSE, P2$\rangle$. This is because MAJ behaves exactly like GSE, except that it locks more sites than GSE (when the transaction is successfully initiated). Similarly, for all predicates P1 and P2, $\langle$MAJ, P1$\rangle$ is compatible with $\langle$TE, P2$\rangle$. This follows from the observation that the quorums of two transactions that are executed using TE and MAJ intersect, so that they cannot execute concurrently, and the allocation constraint is preserved. Thus, a hybrid algorithm, HE, can be constructed that uses MAJ as an intermediate algorithm when a site wishes to switch from GSE to TE: a site first switches from GSE to MAJ and then from MAJ to TE (and vice versa). The switch is performed in such a way that TE and GSE are never used by different sites concurrently.

In HE (Figure 3), site i can be in one of four alg_type states : GSE, MAJ_T, MAJ_G and TE, which each indicate the algorithm that i is currently using. i uses GSE until $Avail_i$ becomes less than a threshold h. At this point, i enters state MAJ_T and starts using MAJ. We assume that in HE, i's current alg_type state is included in each gossip message sent from i. If a site j discovers that site i is in MAJ_T, it enters MAJ_T too, regardless of the value of $Avail_j$. When a site in MAJ_T knows that every other site is also in MAJ_T or TE, it can enter TE. Thus i 'forces' each site to switch to MAJ, and switches to TE when it knows that all sites are using MAJ or TE.

i can switch back to state GSE from state TE when $Avail_i$ becomes greater than or equal to a threshold h'. First i enters MAJ_G and starts using MAJ. Observe that every other site j is either in MAJ_T or in TE when i enters MAJ_G. When j discovers that i has switched to MAJ_G, j also enters MAJ_G (if j is still in MAJ_T, it now knows that since i was in TE, it can also enter TE. It then immediately moves on to MAJ_G). i can transit to GSE when it knows that all sites have entered MAJ_G or GSE. In fact, i can enter GSE when it knows that a majority of sites have entered MAJ_G or GSE (in contrast to the transition from MAJ_T to TE). This is because any site j in TE or MAJ_T will have to consult a majority of sites before it initiates a transaction. and hence will discover that other sites have entered either MAJ_G or GSE. j will change to MAJ_G as a result. Thus when site i wishes to switch to GSE, it 'forces' each other site to also switch to MAJ (and then GSE).

The efficiency of HE depends on the thresholds h and h'. It is desirable that h be chosen as $2 * b$. However, choosing h equal to h' can lead to thrashing, since sites can frequently switch back and forth between GSE and TE. To avoid this, we could introduce some 'hysteresis' by choosing h to be $2 * b - c$ and h' to be $2 * b + c'$, where c and c' are (small) positive integer constants.

Theorem 3 *Algorithm* HE *preserves the allocation constraint.* ■

8.3 Algorithm GSE without locking

We outline a variant of GSE, VGSE, that does not involve locking, but in which more sites may have to be consulted than in GSE before a transaction can be initiated. VGSE does not however suffer from deadlocks as GSE or TE might.

Since quorum sites are not locked, a site can simultaneously be in several quorums. To include another site j in Q_T, i sends a *quorum request* message to j containing $req(T)$. On receiving this message, j logs $req(T)$ as a *requisition* (as opposed to an update u_T where the resources have been granted). j then sends its local site view of updates and requisitions in a *quorum grant* message to i. Let $\mathcal{R}_i$ denote the set of requisitions that i sees after it merges the quorum grant messages. The condition for a transaction to be initiated is now $es_i \geq req(T) + req(\mathcal{U}_i) + req(\mathcal{R}_i)$. Furthermore, suppose the quorums of T and T' intersect at j and the requisition of T' has been included in $\mathcal{R}_i$ (the requisition for T' was sent in the quorum grant message from j). Assume T' was submitted at site k. Then before T can be initiated, i must ensure (by waiting) that either i knows of $u_{T'}$ (so that T' is accounted for

in $Avail_i$) or $req(T)$ has been included in $\mathcal{R}_k$ before T' is initiated (but not both). The requisition for T is upgraded at the quorum sites to the update u_T after RP_T has been executed and the quorum sites informed of u_T. The details of the algorithm and the proof of correctness can be found in [KB92].

The benefit of VGSE over GSE is that a site can be in the quorum of several transactions simultaneously, and can (intuitively) allow those transactions to divide its unused escrow quantity amongst themselves. The drawback of VGSE is the assumption that a requisition is expected to succeed eventually. Thus if i sees a requisition, the escrow that can be used by i is reduced. Hence, site i might be better off including in its quorum a site that is not currently in another quorum. Notice that in GSE, i would have to include only such a site in its quorum, since sites currently in the quorums of other transactions are locked.

9 Conclusions

We have presented the Generalized Site Escrow algorithm which allows decentralized decision-making in a resource allocation application. The algorithm makes use of the semantics of the application and uses the mechanisms of quorum locking and gossip messages to provide a loose synchronization between sites (with respect to the allocations that they have each seen). We have shown that GSE performs well when the quorum size is less than a majority. We have also suggested techniques for optimizing GSE to reduce quorum sizes.

We have thus outlined a suite of algorithms that provide high site autonomy and throughput in executing transactions in a replicated system. We do not provide serializable executions, but show that the allocation constraint is preserved at all times. We believe that concurrency can be improved in certain traditional applications by recasting them as resource allocation applications, and then using techniques described here.

References

[BG91] Barbara, D. and Garcia-Molina, H. *The Demarcation Protocol : A technique for maintaining arithmetic constraints in distributed database systems.* Technical Report CS-TR-320-91, Princeton University, Apr. 1991.

[BHG87] Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publishing Company, 1987.

[Har88] Härder, T. Handling hot spot data in db-sharing systems. *Information Systems*, 13(2):155–166, 1988.

[Her87] Herlihy, M.P. Concurrency vs. availability: atomicity mechanisms for replicated data. *ACM Transactions on Computer Systems*, 5(3):249–274, Aug. 1987.

[HHW89] Heddaya, A., Hsu, M., and Weihl, W.E. Two phase gossip : managing distributed event histories. *Information Sciences*, 49(1):35–57, 1989.

[KB91] Krishnakumar, N. and Bernstein, A.J. Bounded ignorance in replicated systems. In *Proceedings of the 10th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 63–74, 1991. An extended version appears as Technical Report SUSB-TR-90-29 (Oct.1990, revised Aug. 1991), SUNY at Stony Brook.

[KB92] Krishnakumar, N. and Bernstein, A. J. *High Throughput Escrow Algorithms for Replicated Databases.* Technical Report SUSB-TR-92-09, SUNY at Stony Brook, May 1992.

[KS88] Kumar, A. and Stonebraker, M. Semantics based transaction management techniques for replicated data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–388, 1988.

[ONe86] O'Neil, P.E. The escrow transactional model. *ACM Transactions on Database Systems*, 11(4):405–430, Dec. 1986.

[SS90] Soparkar, N. and Silberschatz, A. Data-value partitioning and virtual messages. In *Proceedings of the 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 357–367, 1990.

[WB84] Wuu, G.T.J. and Bernstein, A. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 233–244, 1984.