The Dangers of Replication and a Solution

Jim Gray (Gray@Microsoft.com)
Pat Helland (PHelland@Microsoft.com)
Patrick O'Neil (POneil@cs.UMB.edu)
Dennis Shasha (Shasha@cs.NYU.edu)

Abstract: *Update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up: a ten-fold increase in nodes and traffic gives a thousand fold increase in deadlocks or reconciliations. Master copy replication (primary copy) schemes reduce this problem. A simple analytic model demonstrates these results. A new two-tier replication algorithm is proposed that allows mobile (disconnected) applications to propose tentative update transactions that are later applied to a master copy. Commutative update transactions avoid the instability of other replication schemes.*

1. Introduction

Data is replicated at multiple network nodes for performance and availability. ***Eager replication*** keeps all replicas exactly synchronized at all nodes by updating all the replicas as part of one atomic transaction. Eager replication gives serializable execution – there are no concurrency anomalies. But, eager replication reduces update performance and increases transaction response times because extra updates and messages are added to the transaction.

Eager replication is not an option for mobile applications where most nodes are normally disconnected. Mobile applications require ***lazy replication*** algorithms that asynchronously propagate replica updates to other nodes after the updating transaction commits. Some continuously connected systems use lazy replication to improve response time.

Lazy replication also has shortcomings, the most serious being stale data versions. When two transactions read and write data concurrently, one transaction's updates should be serialized after the other's. This avoids concurrency anomalies. Eager replication typically uses a locking scheme to detect and regulate concurrent execution. Lazy replication schemes typically use a multi-version concurrency control scheme to detect non-serializable behavior [Bernstein, Hadzilacos, Goodman], [Berenson, et. al.]. Most multi-version isolation schemes provide the transaction with the most recent committed value. Lazy replication may allow a transaction to see a very old committed value. Committed updates to a local value may be "in transit" to this node if the update strategy is "lazy".

Eager replication delays or aborts an uncommitted transaction if committing it would violate serialization. Lazy replication has a more difficult task because some replica updates have already been committed when the serialization problem is first detected. There is usually no automatic way to reverse the committed replica updates, rather a program or person must ***reconcile*** conflicting transactions.

To make this tangible, consider a joint checking account you share with your spouse. Suppose it has $1,000 in it. This account is replicated in three places: your checkbook, your spouse's checkbook, and the bank's ledger.

Eager replication assures that all three books have the same account balance. It prevents you and your spouse from writing checks totaling more than $1,000. If you try to overdraw your account, the transaction will fail.

Lazy replication allows both you and your spouse to write checks totaling $1,000 for a total of $2,000 in withdrawals. When these checks arrived at the bank, or when you communicated with your spouse, someone or something reconciles the transactions that used the virtual $1,000.

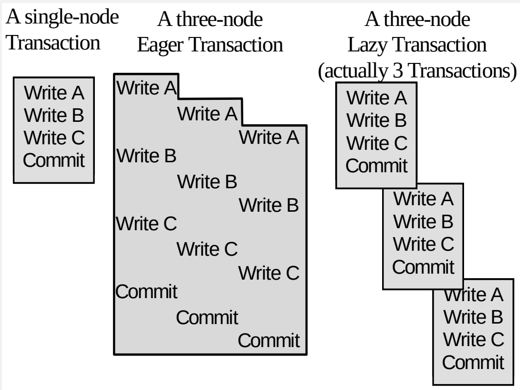
It would be nice to automate this reconciliation. The bank does that by rejecting updates that cause an overdraft. This is a master replication scheme: the bank has the master copy and only the bank's updates really count. Unfortunately, this works only for the bank. You, your spouse, and your creditors are likely to spend considerable time reconciling the "extra" thousand dollars worth of transactions. In the meantime, your books will be inconsistent with the bank's books. That makes it difficult for you to perform further banking operations.

The database for a checking account is a single number, and a log of updates to that number. It is the simplest database. In reality, databases are more complex and the serialization issues are more subtle.

The theme of this paper is that update-anywhere-anytime-anyway replication is unstable.

1. *If the number of checkbooks per account increases by a factor of ten, the deadlock or reconciliation rates rises by a factor of a thousand.*
2. *Disconnected operation and message delays mean lazy replication has more frequent reconciliation.*

Figure 1: When replicated, a simple single-node transaction may apply its updates remotely either as part of the same transaction (*eager*) or as separate transactions (*lazy*). In either case, if data is replicated at *N* nodes, the transaction does *N* times as much work



Simple replication works well at low loads and with a few nodes. This creates a ***scaleup pitfall***. A prototype system demonstrates well. Only a few transactions deadlock or need reconciliation when running on two connected nodes. But the system behaves very differently when the application is scaled up to a large number of nodes, or when nodes are disconnected more often, or when message propagation delays are longer. Such systems have higher transaction rates. Suddenly, the deadlock and reconciliation rate is astronomically higher (cubic growth is predicted by the model). The database at each node diverges further and further from the others as reconciliation fails. Each reconciliation failure implies differences among nodes. Soon, the system suffers ***system delusion*** — the database is inconsistent and there is no obvious way to repair it [Gray & Reuter, pp. 149-150].

This is a bleak picture, but probably accurate. Simple replication (transactional update-anywhere-anytime-anyway) cannot be made to work with global serializability.

In outline, the paper gives a simple model of replication and a closed-form average-case analysis for the probability of waits, deadlocks, and reconciliations. For simplicity, the model ignores many issues that would make the predicted behavior even worse. In particular, it ignores the message propagation delays needed to broadcast replica updates. It ignores "true" serialization, and assumes a weak multi-version form of committed-read serialization (no read locks) [Berenson]. The paper then considers object master replication. Unrestricted lazy master replication has many of the instability problems of eager and group replication.

A restricted form of replication avoids these problems: ***two-tier replication*** has ***base nodes*** that are always connected, and ***mobile nodes*** that are usually disconnected.

1. Mobile nodes propose tentative update transactions to objects owned by other nodes. Each mobile node keeps two object versions: a local version and a best known master version.
2. Mobile nodes occasionally connect to base nodes and propose tentative update transactions to a master node. These proposed transactions are re-executed and may succeed or be rejected. To improve the chances of success, tentative transactions are designed to commute with other transactions. After exchanges the mobile node's database is synchronized with the base nodes. Rejected tentative transactions are reconciled by the mobile node owner who generated the transaction.

Our analysis shows that this scheme supports lazy replication and mobile computing but avoids system delusion: tentative updates may be rejected but the base database state remains consistent.

2. Replication Models

Figure 1 shows two ways to propagate updates to replicas:

1. ***Eager***: Updates are applied to all replicas of an object as part of the original transaction.
2. ***Lazy:*** One replica is updated by the originating transaction. Updates to other replicas propagate asynchronously, typically as a separate transaction for each node.

Figure 2: Updates may be controlled in two ways. Either all updates emanate from a master copy of the object, or updates may emanate from any. Group ownership has many more chances for conflicting updates.

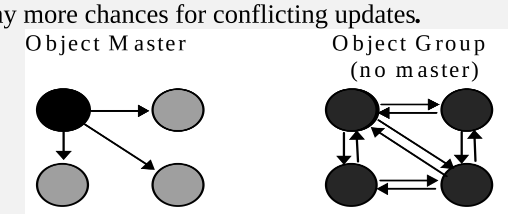


Figure 2 shows two ways to regulate replica updates:

1. ***Group***: Any node with a copy of a data item can update it. This is often called *update anywhere*.
2. ***Master***: Each object has a master node. Only the master can update the *primary copy* of the object. All other replicas are read-only. Other nodes wanting to update the object request the master do the update.

Table 1: A taxonomy of replication strategies contrasting propagation strategy (eager or lazy) with the ownership strategy (master or group).

Propagation vs. Ownership	Lazy	Eager
Group	N transactions N object owners	one transaction N object owners
Master	N transactions one object owner	one transaction one object owner
Two Tier	N+1 transactions, one object owner tentative local updates, eager base updates	

Table 2. Variables used in the model and analysis	
DB_Size	number of distinct objects in the database

Nodes	number of nodes; each node replicates all objects
Transactions	number of concurrent transactions at a node. This is a derived value.
TPS	number of transactions per second originating at this node.
Actions	number of updates in a transaction
Action_Time	time to perform an action
Time_Between_ Disconnects	mean time between network disconnect of a node.
Disconnected_ time	mean time node is disconnected from network
Message_Delay	time between update of an object and update of a replica (ignored)
Message_cpu	processing and transmission time needed to send a replication message or apply a replica update (ignored)

The analysis below indicates that group and lazy replication are more prone to serializability violations than master and eager replication

The model assumes the database consists of a fixed set of objects. There are a fixed number of nodes, each storing a replica of all objects. Each node originates a fixed number of transactions per second. Each transaction updates a fixed number of objects. Access to objects is equi-probable (there are no hotspots). Inserts and deletes are modeled as updates. Reads are ignored. Replica update requests have a transmit delay and also require processing by the sender and receiver. These delays and extra processing are ignored; only the work of sequentially updating the replicas at each node is modeled. Some nodes are mobile and disconnected most of the time. When first connected, a mobile node sends and receives deferred replica updates. Table 2 lists the model parameters.

One can imagine many variations of this model. Applying eager updates in parallel comes to mind. Each design alternative gives slightly different results. The design here roughly characterizes the basic alternatives. We believe obvious variations will not substantially change the results here.

Each node generates *TPS* transactions per second. Each transaction involves a fixed number of actions. Each action requires a fixed time to execute. So, a transaction's duration is *Actions x Action_Time*. Given these two observations, the number of concurrent transactions originating at a node is:

$$Transactions = TPS \times Actions \times Action\_Time \qquad (1)$$

A more careful analysis would consider that fact that, as system load and contention rises, the time to complete an action increases. In a scaleable server system, this *time-dilation* is a second-order effect and is ignored here.

In a system of *N* nodes, *N* times as many transactions will be originating per second. Since each update transaction must replicate its updates to the other *(N-1)* nodes, it is easy to see

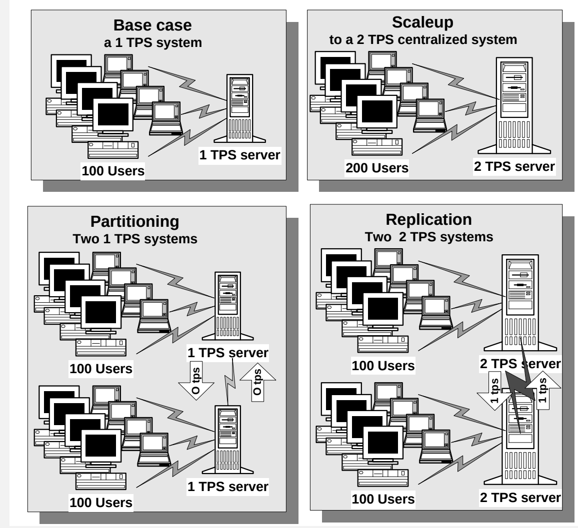


Figure 3: Systems can grow by (1) *scaleup*: buying a bigger machine, (2) *partitioning*: dividing the work between two machines, or (3) *replication*: placing the data at two machines and having each machine keep the data current. This simple idea is key to understanding the N^2 growth. Notice that each of the replicated servers at the lower right of the illustration is performing 2 TPS and the aggregate rate is 4 TPS. Doubling the users increased the total workload by a factor of four. Read-only transactions need not generate any additional load on remote nodes.

that the transaction size for eager systems grows by a factor of *N* and the node update rate grows by N^2. In lazy systems, each *user* update transaction generates *N-1* lazy replica updates, so there are *N* times as many concurrent transactions, and the node update rate is N^2 higher. This non-linear growth in node update rates leads to unstable behavior as the system is scaled up.

3. Eager Replication

Eager replication updates all replicas when a transaction updates any instance of the object. There are no serialization anomalies (inconsistencies) and no need for reconciliation in eager systems. Locking detects potential anomalies and converts them to waits or deadlocks.

With eager replication, reads at connected nodes give current data. Reads at disconnected nodes may give stale (out of date) data. Simple eager replication systems prohibit updates if any node is disconnected. For high availability, eager replication systems allow updates among members of the quorum or cluster [Gifford], [Garcia-Molina]. When a node joins the quorum, the quorum sends the new node all replica updates since the node was disconnected. We assume here that a quorum or fault tolerance scheme is used to improve update availability. Even if all the nodes are connected all the time, updates may fail due to deadlocks that prevent serialization errors. The following simple analysis derives the wait and deadlock rates of an eager replication system. We start with wait and deadlock rates for a single-node system.

In a single-node system the "other" transactions have about $\frac{Tranascations \times Actions}{2}$ resources locked (each is about half way complete). Since objects are chosen uniformly from the database, the chance that a request by one transaction will request a resource locked by any other transaction is: $\frac{Transactions \times Actions}{2 \times DB\_size}$. A transaction makes *Actions* such requests, so the chance that it will wait sometime in its lifetime is approximately [Gray et. al.], [Gray & Reuter pp. 428]:

$$PW \approx 1-(1-\frac{Transactions \times Actions}{2 \times DB\_size})^{Acctions} \approx \frac{Transactions \times Actions^2}{2 \times DB\_Size} \quad (2)$$

A deadlock consists of a cycle of transactions waiting for one another. The probability a transaction forms a cycle of length two is PW^2 divided by the number of transactions. Cycles of length j are proportional to PW^j and so are even less likely if $PW << 1$. Applying equation (1), the probability that the transaction deadlocks is approximately:

$$PD \approx \frac{PW^2}{Transactions} \frac{Transactions \times Actions^4}{4 \times DB\_Size^2} = \frac{TPS \times Action\_Time \times Actions^5}{4 \times DB\_Size^2} \quad (3)$$

Equation (3) gives the deadlock hazard for a transaction. The deadlock rate for a transaction is the probability it deadlock's in the next second. That is PD divided by the transaction lifetime (*Actions x Action_Time*).

$$Trans\_Deadlock\_rate \approx \frac{TPS \times Actions^4}{4 \times DB\_Size^2} \quad (4)$$

Since the node runs *Transactions* concurrent transactions, the deadlock rate for the whole node is higher. Multiplying equation (4) and equation (1), the node deadlock rate is:

$$Node\_Deadlock\_Rate \approx \frac{TPS^2 \times Action\_Time \times Actions^5}{4 \times DB\_Size^2} \quad (5)$$

Suppose now that several such systems are replicated using eager replication — the updates are done immediately as in Figure 1. Each node will initiate its local load of *TPS* transactions per second[1]. The transaction size, duration, and aggregate transaction rate for eager systems is:

Transaction_Size = Actions x Nodes
Transaction_Duration = Actions x Nodes x Action_Time
Total_TPS = TPS x Nodes *(6)*

Each node is now doing its own work and also applying the updates generated by other nodes. So each update transaction actually performs many more actions (*Nodes x Actions*) and so has a much longer lifetime — indeed it takes at least *Nodes* times longer[2]. As a result the total number of transactions in the system rises quadratically with the number of nodes:

Total_Transactions = TPS x Actions x Action_Time x $Nodes^2$ *(7)*

This rise in active transactions is due to eager transactions taking *N*-Times longer and due to lazy updates generating *N*-times more transactions. The action rate also rises very fast with *N*. Each node generates work for all other nodes. The eager work rate, measured in actions per second is:

Action_Rate = Total_TPS x Transaction_Size
= TPS x Actions x $Nodes^2$ *(8)*

It is surprising that the action rate and the number of active transactions is the same for eager and lazy systems. Eager systems have fewer-longer transactions. Lazy systems have more and shorter transactions. So, although equations (6) are different for lazy systems, equations (7) and (8) apply to both eager and lazy systems.

Ignoring message handling, the probability a transaction waits can be computed using the argument for equation (2). The transaction makes *Actions* requests while the other Total_Transactions have *Actions/2* objects locked. The result is approximately:

$$PW\_eager \approx Total\_Transactions \times Actions \times \frac{Actions}{2 \times DB\_Size}$$
$$= \frac{TPS \times Action\_Time \times Actions^3 \times Nodes^2}{2 \times DB\_Size} \quad (9)$$

This is the probability that one transaction waits. The wait rate (waits per second) for the entire system is computed as:

$$Total\_Eager\_Wait\_Rate$$
$$\approx \frac{PW\_eager}{Transaction\_Duration} \times Total\_Transactions \quad (10)$$
$$= \frac{TPS^2 \times Action\_Time \times (Actions \times Nodes)^3}{2 \times DB\_Size}$$

As with equation (4), The probability that a particular transaction deadlocks is approximately:

$$PD\_eager \approx \frac{Total\_Transactions \times Actions^4}{4 \times DB\_Size^2}$$
$$= \frac{TPS \times Action\_Time \times Actions^5 \times Nodes^2}{4 \times DB\_Size^2} \quad (11)$$

The equation for a single-transaction deadlock implies the total deadlock rate. Using the arguments for equations (4) and (5), and using equations (7) and (11):

[1] The assumption that transaction arrival rate per node stays constant as nodes are replicated assumes that nodes are lightly loaded. As the replication workload increases, the nodes must grow processing and IO power to handle the increased load. Growing power at an N^2 rate is problematic.

[2] An alternate model has eager actions broadcast the update to all replicas in one instant. The replicas are updated in parallel and the elapsed time for each action is constant (independent of *N*). In our model, we attempt to capture message handing costs by serializing the individual updates. If one follows this model, then the processing at each node rises quadraticly, but the number of concurrent transactions stays constant with scaleup. This model avoids the polynomial explosion of waits and deadlocks if the total TPS rate is held constant.

$$Total\_Eager\_Deadlock\_Rate$$
$$\approx Total\_Transactions \times \frac{PD\_eager}{Transaction\_Duration} \quad (12)$$
$$\approx \frac{TPS^2 \times Action\_Time \times Actions^5 \times Nodes^3}{4 \times DB\_Size^2}$$

If message delays were added to the model, then each transaction would last much longer, would hold resources much longer, and so would be more likely to collide with other transactions. Equation (12) also ignores the "second order" effect of two transactions racing to update the same object at the same time (it does not distinguish between *Master* and *Group* replication). If *DB_Size >> Node,* such conflicts will be rare.

This analysis points to some serious problems with eager replication. Deadlocks rise as the third power of the number of nodes in the network, and the fifth power of the transaction size. Going from one-node to ten nodes increases the deadlock rate a thousand fold. A ten-fold increase in the transaction size increases the deadlock rate by a factor of 100,000.

To ameliorate this, one might imagine that the database size grows with the number of nodes (as in the checkbook example earlier, or in the TPC-A, TPC-B, and TPC-C benchmarks). More nodes, and more transactions mean more data. With a scaled up database size, equation (12) becomes:

$$Eager\_Deadlock\_Rate\_Scaled\_DB$$
$$\approx \frac{TPS^2 \times Action\_Time \times Actions^5 \times Nodes}{4 \times DB\_Size^2} \quad (13)$$

Now a ten-fold growth in the number of nodes creates *only* a ten-fold growth in the deadlock rate. This is still an unstable situation, but it is a big improvement over equation (12)

Having a master for each object helps eager replication avoid deadlocks. Suppose each object has an owner node. Updates go to this node first and are then applied to the replicas. If, each transaction updated a single replica, the object-master approach would eliminate all deadlocks.

In summary, eager replication has two major problems:

1. Mobile nodes cannot use an eager scheme when disconnected.
2. The probability of deadlocks, and consequently failed transactions rises very quickly with transaction size and with the number of nodes. A ten-fold increase in nodes gives a thousand-fold increase in failed transactions (deadlocks).

We see no solution to this problem. If replica updates were done concurrently, the action time would not increase with *N* then the growth rate would *only* be quadratic.

4. Lazy Group Replication

Lazy group replication allows any node to update any local data. When the transaction commits, a transaction is sent to every other node to apply the root transaction's updates to the replicas at the destination node (see Figure 4). It is possible for two nodes to update the same object and race each other to install their updates at other nodes. The replication mechanism must detect this and reconcile the two transactions so that their updates are not lost.

Timestamps are commonly used to detect and reconcile lazy-group transactional updates. Each object carries the timestamp of its most recent update. Each replica update carries the new value and is tagged with the old object timestamp. Each node detects incoming replica updates that would overwrite earlier committed updates. The node tests if the local replica's timestamp and the update's old timestamp are equal. If so, the update is safe. The local replica's timestamp advances to the new transaction's timestamp and the object value is updated. If the current timestamp of the local replica does not match the old timestamp seen by the root transaction, then the update may be "dangerous". In such cases, the node rejects the incoming transaction and submits it for *reconciliation*.

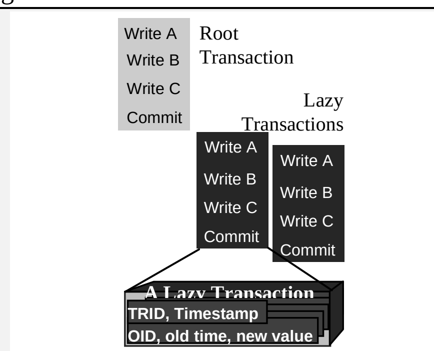


Figure 4: A lazy transaction has a root execution that updates either master or local copies of data. Then subsequent transactions update replicas at remote nodes — one lazy transaction per remote replica node. The lazy updates carry timestamps of each original object. If the local object timestamp does not match, the update may be dangerous and some form of reconciliation is needed.

Transactions that would wait in an eager replication system face reconciliation in a lazy-group replication system. Waits are much more frequent than deadlocks because it takes two waits to make a deadlock. Indeed, if waits are a rare event, then deadlocks are very rare *(rare²)*. Eager replication waits cause delays while deadlocks create application faults. With lazy replication, the much more frequent waits are what determines the reconciliation frequency. So, the system-wide lazy-group reconciliation rate follows the transaction wait rate equation (Equation 10):

$$Lazy\_Group\_Reconciliation\_Rate \approx \frac{TPS^2 \times Action\_Time \times (Actions \times Nodes)^3}{2 \times DB\_Size} \quad (14)$$

As with eager replication, if message propagation times were added, the reconciliation rate would rise. Still, having the reconciliation rate rise by a factor of a thousand when the system scales up by a factor of ten is frightening.

The really bad case arises in mobile computing. Suppose that the typical node is disconnected most of the time. The node accepts and applies transactions for a day. Then, at night it connects and downloads them to the rest of the network. At that time it also accepts replica updates. It is as though the message propagation time was 24 hours.

If any two transactions at any two different nodes update the same data during the disconnection period, then they will need reconciliation. What is the chance of two disconnected transactions colliding during the *Disconnected_Time*?

If each node updates a small fraction of the database each day then the number of distinct *outbound* pending object updates at reconnect is approximately:

$$Outbound\_Updates \approx Disconnect\_Time \times TPS \times Actions \quad (15)$$

Each of these updates applies to all the replicas of an object. The pending *inbound updates* for this node from the rest of the network is approximately *(Nodes-1)* times larger than this.

$$Inbound\_Updates \approx (Nodes - 1) \times Disconnect\_Time \times TPS \times Actions \quad (16)$$

If the inbound and outbound sets overlap, then reconciliation is needed. The chance of an object being in both sets is approximately:

$$P(collision) \approx \frac{Inbound\_Updates \times Outbound\_Updates}{DB\_Size} \approx \frac{Nodes \times (Disconnect\_Time \times TPS \times Actions)^2}{DB\_Size} \quad (17)$$

Equation (17) is the chance one node needs reconciliation during the *Disconnect_Time* cycle. The rate for all nodes is:

$$Lazy\_Group\_Reconciliation\_Rate \approx P(collision) \times \frac{Nodes}{Disconnect\_Time} \approx \frac{Disconnect\_Time \times (TPS \times Actions \times Nodes)^2}{DB\_Size} \quad (18)$$

The quadratic nature of this equation suggests that a system that performs well on a few nodes with simple transactions may become unstable as the system scales up.

5. Lazy Master Replication

Master replication assigns an owner to each object. The owner stores the object's correct current value. Updates are first done by the owner and then propagated to other replicas. Different objects may have different owners.

When a transaction wants to update an object, it sends an RPC (remote procedure call) to the node owning the object. To get serializability, a read action should send read-lock RPCs to the masters of any objects it reads.

To simplify the analysis, we assume the node originating the transaction broadcasts the replica updates to all the slave replicas after the master transaction commits. The originating node sends one slave transaction to each slave node (as in Figure 1). Slave updates are timestamped to assure that all the replicas converge to the same final state. If the record timestamp is newer than a replica update timestamp, the update is "stale" and can be ignored. Alternatively, each master node sends replica updates to slaves in sequential commit order.

Lazy-Master replication is not appropriate for mobile applications. A node wanting to update an object must be connected to the object owner and participate in an atomic transaction with the owner.

As with eager systems, lazy-master systems have no reconciliation failures; rather, conflicts are resolved by waiting or deadlock. Ignoring message delays, the deadlock rate for a lazy-master replication system is similar to a single node system with much higher transaction rates. Lazy master transactions operate on master copies of objects. But, because there are *Nodes* times more users, there are *Nodes* times as many concurrent master transactions and approximately $Nodes^2$ times as many replica update transactions. The replica update transactions do not really matter, they are background housekeeping transactions. They can abort and restart without affecting the user. So the main issue is how frequently the master transactions deadlock. Using the logic of equation (5), the deadlock rate is approximated by:

$$Lazy\_Master\_Deadlock\_Rate \approx \frac{(TPS \times Nodes)^2 \times Action\_Time \times Actions^5}{4 \times DB\_Size^2} \quad (19)$$

This is better behavior than lazy-group replication. Lazy-master replication sends fewer messages during the base transaction and so completes more quickly. Nevertheless, all of these replication schemes have troubling deadlock or reconciliation rates as they grow to many nodes.

In summary, lazy-master replication requires contact with object masters and so is not useable by mobile applications. Lazy-master replication is slightly less deadlock prone than eager-group replication primarily because the transactions have shorter duration.

6. Non-Transactional Replication Schemes

The equations in the previous sections are facts of nature — they help explain another fact of nature. They show why there are no high-update-traffic replicated databases with globally serializable transactions.

Certainly, there are replicated databases: bibles, phone books, check books, mail systems, name servers, and so on. But updates to these databases are managed in interesting ways — typically in a lazy-master way. Further, updates are not record-value oriented; rather, updates are expressed as transactional transformations such as "Debit the account by $50" instead of "change account from $200 to $150".

One strategy is to abandon serializabilty for the ***convergence property***: if no new transactions arrive, and if all the nodes are connected together, they will all converge to the same replicated state after exchanging replica updates. The resulting state contains the committed appends, and the most recent replacements, but updates may be lost.

Lotus Notes gives a good example of convergence [Kawell]. Notes is a lazy group replication design (update anywhere, anytime, anyhow). Notes provides convergence rather than an ACID transaction execution model. The database state may not reflect any particular serial execution, but all the states will be identical. As explained below, timestamp schemes have the lost-update problem.

Lotus Notes achieves convergence by offering lazy-group replication at the transaction level. It provides two forms of update transaction:

1. ***Append*** adds data to a Notes file. Every appended note has a timestamp. Notes are stored in timestamp order. If all nodes are in contact with all others, then they will all converge on the same state.
2. ***Timestamped replace a value*** replaces a value with a newer value. If the current value of the object already has a timestamp greater than this update's timestamp, the incoming update is discarded.

If convergence were the only goal, the timestamp method would be sufficient. But, the timestamp scheme may lose the effects of some transactions because it just applies the most recent updates. Applying a timestamp scheme to the checkbook example, if there are two concurrent updates to a checkbook balance, the highest timestamp value wins and the other update is discarded as a "stale" value. Concurrency control theory calls this the *lost update problem*. Timestamp schemes are vulnerable to lost updates.

Convergence is desirable, but the converged state should reflect the effects of all committed transactions. In general this is not possible unless global serialization techniques are used.

In certain cases transactions can be designed to commute, so that the database ends up in the same state no matter what transaction execution order is chosen. Timestamped Append is a kind of commutative update but there are others (e.g., adding and subtracting constants from an integer value). It would be possible for Notes to support a third form of transaction:

3. ***Commutative updates*** that are incremental transformations of a value that can be applied in any order.

Lotus Notes, the Internet name service, mail systems, Microsoft Access, and many other applications use some of these techniques to achieve convergence and avoid delusion.

Microsoft Access offers convergence as follows. It has a single design master node that controls all schema updates to a replicated database. It offers update-anywhere for record instances. Each node keeps a version vector with each replicated record. These version vectors are exchanged on demand or periodically. The most recent update wins each pairwise exchange. Rejected updates are reported [Hammond].

The examples contrast with a simple update-anywhere-anytime-anyhow lazy-group replication offered by some systems. If the transaction profiles are not constrained, lazy-group schemes suffer from unstable reconciliation described in earlier sections. Such systems degenerate into system delusion as they scale up.

Lazy group replication schemes are emerging with specialized reconciliation rules. Oracle 7 provides a choice of twelve reconciliation rules to merge conflicting updates [Oracle]. In addition, users can program their own reconciliation rules. These rules give priority certain sites, or time priority, or value priority, or they merge commutative updates. The rules make some transactions commutative. A similar, transaction-level approach is followed in the two-tier scheme described next.

7. Two-Tier Replication

An ideal replication scheme would achieve four goals:

Availability and scaleability: Provide high availability and scaleability through replication, while avoiding instability.

Mobility: Allow mobile nodes to read and update the database while disconnected from the network.

Serializability: Provide single-copy serializable transaction execution.

Convergence: Provide convergence to avoid system delusion.

The safest transactional replication schemes, (ones that avoid system delusion) are the eager systems and lazy master systems. They have no reconciliation problems (they have no reconciliation). But these systems have other problems. As shown earlier:

1. Mastered objects cannot accept updates if the master node is not accessible. This makes it difficult to use master replication for mobile applications.
2. Master systems are unstable under increasing load. Deadlocks rise quickly as nodes are added.

3. Only eager systems and lazy master (where reads go to the master) give ACID serializability.

Circumventing these problems requires changing the way the system is used. We believe a scaleable replication system must function more like the check books, phone books, Lotus Notes, Access, and other replication systems we see about us.

Lazy-group replication systems are prone to reconciliation problems as they scale up. Manually reconciling conflicting transactions is unworkable. One approach is to *undo* all the work of any transaction that needs reconciliation — backing out all the updates of the transaction. This makes transactions atomic, consistent, and isolated, but not durable — or at least not durable until the updates are propagated to each node. In such a lazy group system, every transaction is tentative until all its replica updates have been propagated. If some mobile replica node is disconnected for a very long time, all transactions will be tentative until the missing node reconnects. So, an undo-oriented lazy-group replication scheme is untenable for mobile applications.

The solution seems to require a modified mastered replication scheme. To avoid reconciliation, each object is mastered by a node — much as the bank owns your checking account and your mail server owns your mailbox. Mobile agents can make tentative updates, then connect to the base nodes and immediately learn if the tentative update is acceptable.

The ***two-tier replication*** scheme begins by assuming there are two kinds of nodes:

Mobile nodes are disconnected much of the time. They store a replica of the database and may originate tentative transactions. A mobile node may be the master of some data items.

Base nodes are always connected. They store a replica of the database. Most items are mastered at base nodes.

Replicated data items have two versions at mobile nodes:

Master Version: The most recent value received from the object master. The version at the object master is *the* master version, but disconnected or lazy replica nodes may have older versions.

Tentative Version: The local object may be updated by tentative transactions. The most recent value due to local updates is maintained as a tentative value.

Similarly, there are two kinds of transactions:

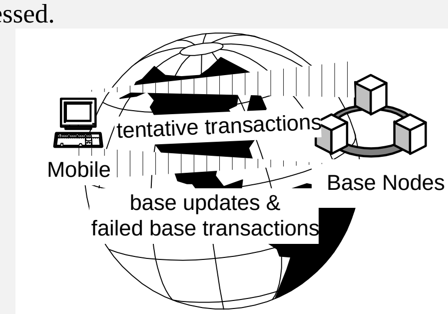
Base Transaction: Base transactions work only on master data, and they produce new master data. They involve at most one connected-mobile node and may involve several base nodes.

Tentative Transaction: Tentative transactions work on local tentative data. They produce new tentative versions. They also produce a base transaction to be run at a later time on the base nodes.

Tentative transactions must follow a ***scope rule***: they may involve objects mastered on base nodes and mastered at the mobile node originating the transaction (call this the transaction's *scope*). The idea is that the mobile node and all the base nodes will be in contact when the tentative transaction is processed as a "real" base transaction — so the real transaction will be able to read the master copy of each item in the scope.

Local transactions that read and write *only* local data can be designed in any way you like. They cannot read-or write any tentative data because that would make them tentative.

Figure 5: The two-tier-replication scheme. Base nodes store replicas of the database. Each object is mastered at some node. Mobile nodes store a replica of the database, but are usually disconnected. Mobile nodes accumulate tentative transactions that run against the tentative database stored at the node. Tentative transactions are reprocessed as base transactions when the mobile node reconnects to the base. Tentative transactions may fail when reprocessed.



The base transaction generated by a tentative transaction may fail or it may produce different results. The base transaction has an ***acceptance criterion***: a test the resulting outputs must pass for the slightly different base transaction results to be acceptable. To give some sample acceptance criteria:

- The bank balance must not go negative.
- The price quote can not exceed the tentative quote.
- The seats must be aisle seats.

If a tentative transaction fails, the originating node and person who generated the transaction are informed it failed and why it failed. Acceptance failure is equivalent to the reconciliation mechanism of the lazy-group replication schemes. The differences are (1) the master database is always converged — there is no system delusion, and (2) the originating node need only contact a base node in order to discover if a tentative transaction is acceptable.

To continue the checking account analogy, the bank's version of the account is the master version. In writing checks, you and your spouse are creating tentative transactions which result in tentative versions of the account. The bank runs a base transaction when it clears the check.

If you contact your bank and it clears the check, then you know the tentative transaction is a real transaction.

Consider the two-tier replication scheme's behavior during connected operation. In this environment, a two-tier system operates much like a lazy-master system with the additional restriction that no transaction can update data mastered at more than one mobile node. This restriction is not really needed in the connected case.

Now consider the disconnected case. Imagine that a mobile node disconnected a day ago. It has a copy of the base data as of yesterday. It has generated tentative transactions on that base data and on the local data mastered by the mobile node. These transactions generated tentative data versions at the mobile node. If the mobile node queries this data it sees the tentative values. For example, if it updated documents, produced contracts, and sent mail messages, those tentative updates are all visible at the mobile node.

When a mobile node connects to a base node, the mobile node:
1. Discards its tentative object versions since they will soon be refreshed from the masters,
2. Sends replica updates for any objects mastered at the mobile node to the base node "hosting" the mobile node,
3. Sends all its tentative transactions (and all their input parameters) to the base node to be executed in the order in which they committed on the mobile node,
4. Accepts replica updates from the base node (this is standard lazy-master replication), and
5. Accepts notice of the success or failure of each tentative transaction.

The "host" base node is the other tier of the two tiers. When contacted by a mobile note, the host base node:
1. Sends delayed replica update transactions to the mobile node.
2. Accepts delayed update transactions for mobile-mastered objects from the mobile node.
3. Accepts the list of tentative transactions, their input messages, and their acceptance criteria. Reruns each tentative transaction in the order it committed on the mobile node. During this reprocessing, the base transaction reads and writes object master copies using a lazy-master execution model. The scope-rule assures that the base transaction only accesses data mastered by the originating mobile node and base nodes. So master copies of all data in the transaction's scope are available to the base transaction. If the base transaction fails its acceptance criteria, the base transaction is aborted and a diagnostic message is returned to the mobile node. If the acceptance criteria requires the base and tentative transaction have identical outputs, then subsequent transactions reading tentative results written by T will fail too. On the other hand, weaker acceptance criteria are possible.
4. After the base node commits a base transaction, it propagates the lazy replica updates as transactions sent to all the other replica nodes. This is standard lazy-master.
5. When all the tentative transactions have been reprocessed as base transactions, the mobile node's state is converged with the base state.

The key properties of the two-tier replication scheme are:
1. Mobile nodes may make tentative database updates.
2. Base transactions execute with single-copy serializability so the master base system state is the result of a serializable execution.
3. A transaction becomes durable when the base transaction completes.
4. Replicas at all connected nodes converge to the base system state.
5. If all transactions commute, there are no reconciliations.

This comes close to meeting the four goals outlined at the start of this section.

When executing a base transaction, the two-tier scheme is a lazy-master scheme. So, the deadlock rate for base transactions is given by equation (19). This is still an N^2 deadlock rate. If a base transaction deadlocks, it is resubmitted and reprocessed until it succeeds, much as the replica update transactions are resubmitted in case of deadlock.

The reconciliation rate for base transactions will be zero if all the transactions commute. The reconciliation rate is driven by the rate at which the base transactions fail their acceptance criteria.

Processing the base transaction may produce results different from the tentative results. This is acceptable for some applications. It is fine if the checking account balance is different when the transaction is reprocessed. Other transactions from other nodes may have affected the account while the mobile node was disconnected. But, there are cases where the changes may not be acceptable. If the price of an item has increased by a large amount, if the item is out of stock, or if aisle seats are no longer available, then the salesman's price or delivery quote must be reconciled with the customer.

These acceptance criteria are application specific. The replication system can do no more than detect that there is a difference between the tentative and base transaction. This is probably too pessimistic a test. So, the replication system will simply run the tentative transaction. If the tentative transaction completes successfully and passes the acceptance test, then the replication system assumes all is well and propagates the replica updates as usual.
Users are aware that all updates are tentative until the transaction becomes a base transaction. If the base transaction fails, the user may have to revise and resubmit a transaction. The programmer must design the transactions to be commutative and to have acceptance criteria to de-

tect whether the tentative transaction agrees with the base transaction effects.

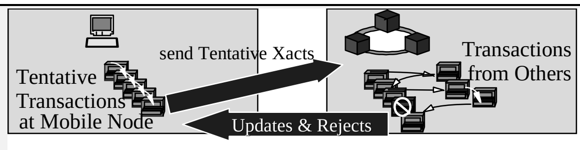


Figure 6: Executing tentative and base transactions in two-tier replication.

Thinking again of the checkbook example of an earlier section. The check is in fact a tentative update being sent to the bank. The bank either honors the check or rejects it. Analogous mechanisms are found in forms flow systems ranging from tax filing, applying for a job, or subscribing to a magazine. It is an approach widely used in human commerce.

This approach is similar to, but more general than the Data Cycle architecture [Herman] which has a single master node for all objects.

The approach can be used to obtain pure serializability if the base transaction only reads and writes master objects (current versions).

8. Summary

Replicating data at many nodes and letting anyone update the data is problematic. Security is one issue, performance is another. When the standard transaction model is applied to a replicated database, the size of each transaction rises by the degree of replication. This, combined with higher transaction rates means dramatically higher deadlock rates.

It might seem at first that a lazy replication scheme will solve this problem. Unfortunately, lazy-group replication just converts waits and deadlocks into reconciliations. Lazy-master replication has slightly better behavior than eager-master replication. Both suffer from dramatically increased deadlock as the replication degree rises. None of the master schemes allow mobile computers to update the database while disconnected from the system.

The solution appears to be to use semantic tricks (timestamps, and commutative transactions), combined with a two-tier replication scheme. Two-tier replication supports mobile nodes and combines the benefits of an eager-master-replication scheme and a local update scheme.

9. Acknowledgments

Tanj (John G.) Bennett of Microsoft and Alex Thomasian of IBM gave some very helpful advice on an earlier version of this paper. The anonymous referees made several helpful suggestions to improve the presentation. Dwight Joe pointed out a mistake in the published version of equation 19.

10. References

Bernstein, P.A., V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison Wesley, Reading MA., 1987.

Berenson, H., Bernstein, P.A., Gray, J., Jim Melton, J., O'Neil, E., O'Neil, P., "A Critique of ANSI SQL Isolation Levels," Proc. ACM SIGMOD 95, pp. 1-10, San Jose CA, June 1995.

Garcia Molina, H. "Performance of Update Algorithms for Replicated Data in a Distributed Database," TR STAN-CS-79-744, CS Dept., Stanford U., Stanford, CA., June 1979.

Garcia Molina, H., Barbara, D., "How to Assign Votes in a Distributed System," J. ACM, 32(4). Pp. 841-860, October, 1985.

Gifford, D. K., "Weighted Voting for Replicated Data," Proc. ACM SIGOPS SOSP, pp: 150-159, Pacific Grove, CA, December 1979.

Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann, San Francisco, CA. 1993.

Gray, J., Homan, P, Korth, H., Obermarck, R., "A Strawman Analysis of the Probability of Deadlock," IBM RJ 2131, IBM Research, San Jose, CA., 1981.

Hammond, Brad, "Wingman, A Replication Service for Microsoft Access and Visual Basic", Microsoft White Paper, bradha@microsoft.com

Herman, G., Gopal, G, Lee, K., Weinrib, A., "The Datacycle Architecture for Very High Throughput Database Systems," Proc. ACM SIGMOD, San Francisco, CA. May 1987.

Kawell, L.., Beckhardt, S., Halvorsen, T., Raymond Ozzie, R., Greif, I.,"Replicated Document Management in a Group Communication System," Proc. Second Conference on Computer Supported Cooperative Work, Sept. 1988.

Oracle, "Oracle7 Server Distributed Systems: Replicated Data," Oracle part number A21903.March 1994, Oracle, Redwood Shores, CA. Or http://www.oracle.com/products/oracle7/server/whitepapers/replication/html/index