A Transaction Replication Scheme for a Replicated Database with Node Autonomy

Ada Wai-chee Fu
Chinese University of Hong Kong
adafu@cs.cuhk.hk

David Wai-Lok Cheung
The University of Hong Kong
dcheung@csd.hku.hk

Abstract

Many proposed protocols for replicated databases consider centralized control of each transaction so that given a transaction, some site will monitor the remote data access and transaction commit. We consider the approach of broadcasting transactions to remote sites and handling these transactions in their complete form at each site. We consider data of two types: shared-private data and public data and show that transactions working only on shared-private data can be executed under a local concurrency control protocol. We assume a synchronized network with possibilities of partition failures. We show that in our scheme transaction execution can be managed with less communication delay compared to centralized transaction control.

1 Motivation

Many replicated distributed database protocols manage the execution of a given transaction at one site, accessing local and remote data copies for its operations, and organize the commit or abort of the transaction from that site. We refer to this as *centralized transaction management.* With centralized transaction management, when there are a number of operations $o_1, o_2, o_3, ...o_n$ in a transaction and some operation o_i depends on the result of some previous operation o_j, some communication overhead is required to secure the result of o_j before it can begin o_i. To eliminate this type of overhead a transaction must be executed entirely at one site which contains replication of all relevant data so that the execution can determine the outcome of each operation locally at the site. One possible way to achieve this is by replication of data and transactions, meaning that execution of entire transactions are replicated at the data replication sites. This paper deals with this possibility.

In some distributed database applications, users may be able to distinguish two types of data. The first type of logical data object is *owned by* (*private to*) a particular site, meaning that only transactions submitted from that site can modify the logical data object, and moreover, a transaction that modifies the data object would not need to read data from other parts of the network. Transactions submitted from other sites may read (*share*) this type of data. We call this type of data **shared-private** data. The second type of logical data is **public**, meaning that these logical data objects can be read and modified by transactions submitted from every site.

As an example, consider an airline database system. There may be a site for accounting, a site for flight scheduling, and many sites for seat reservation. The accounting and flight scheduling data would be owned by the accounting site and the flight scheduling site, respectively. These data may depend in some way on the seat reservation data, for example, we may determine future flight schedules based on previous flights' consumer pattern, however, we do not need to read seat plans and write to flight schedules within a single transaction. The reservation sites may want to read flight schedules and policies determined by the accounting site but they will not modify them. Hence flight schedules and accounting policies are shared-private data. The seat plan of each flight should be public among the reservation sites since each such site

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

can book seats and modify the data. Hence the seat plans are considered public data.

Since shared-private data is updated by only one site, we might expect that simpler concurrency control is sufficient. In our scheme, transactions accessing only shared-private data can normally be executed and committed under a local concurrency control protocol with little interaction with other sites. To our knowledge, most previous work on replicated distributed database systems assume only public data, so transactions from any site can operate on any data object. With shared-private data, we utilize the semantics of node autonomy. In [KoG87] data can be updated by only one node and can be read by other sites according to a read-access graph. More arguments that support the concept of node autonomy can be found in [CS80], [LS80], [GaK88].

We shall adopt one-copy serializability (see section 4) to be our correctness criteria. When transactions are replicated at many sites, a main problem is to make sure that the essential ("serialization") orderings of transaction execution are identical at all the replication sites. A locking mechanism which dynamically assigns the ordering would not work. One possible way is to make use of global timestamps. In order to get good performance, timestamps based on closely synchronized clocks are used. We must also control the dynamic transaction aborts of timestamping protocol. This is achievable by a conservative timestamping method, which requires that user transactions predeclare supersets of data they read/write. Although this would eliminate aborts in normal operation, we must still consider transaction aborts due to exceptional situations (e.g. a site running out of disk space or a partition failure). This is handled by the commit protocol. In the case of partition failure, we would not want the system to fail totally, and hence a partition failure protocol is used. We propose a replication control scheme based on replicated transaction processing. We shall call it the **Transaction Replication Scheme (TRS)**. A conservative timestamping approach is followed. We consider fail-stop failures and partition failures. The basic ideas of decentralized two-phase commit protocol ([Ske82]) and virtual partition protocol ([ET86], [ET89]) are adopted. Multiple versions of shared-private data may also be necessary.

The performance of the system would depend on the accuracy of clock synchronization. In particular, the response time of public transactions and the number of versions of shared-private data that may be needed depend on clock synchronization. We quote from [Lis91]: "*Synchronized clocks are quickly becoming a reality in distributed systems. For example, the network time protocol NTP [Mil88] synchronizes clocks of nodes on geographically distributed networks. It does this at low cost and provides clocks that are synchronized to within a few milliseconds of one another. NTP is running on the internet today and is used to synchronize clocks of nodes throughout the United States, Canada, and various places in Europe.*" Hence we believe that clock synchronization is a valid assumption. If indeed we cannot rely on an existing clock synchronization system, logical clocks can be created as in [Lam78].

Replicated transaction processing has been investigated in [PiG89], where protocols of high reliability by comparing results of transaction replicas are proposed. Our work here is more concerned about the performance. Other work that studies performance of replicated databases include [TT91] where transaction management is "decentralized" somewhat by having a "leader replica of data" which co-ordinates the accesses to other copies, but the communication overheads described above still exist.

There are 2 main findings in this paper. First, we show that with clock synchronization, transaction replication can help to reduce communication overheads in the transaction management of distributed databases. Second, we show that by dividing data into 2 types, "public" and "shared-private", instead of treating all data as "public", we can adopt local concurrency control for transactions that work only on shared-private data and also enhance their performance. The rest of the paper is organized as follows. We shall first describe the system model we assume. Then we describe the basic operations of synchronized transaction broadcasting. Next we describe a solution to global commit and to handle partition failure. Finally we give a correctness argument and a conclusion.

2 Network Model

Our system model consists of a set of processing *sites* (or *nodes*) connected through a communication *network*. All processing needed by distributed applications is performed at sites, while any processing needed for communication (e.g. routing) is performed by the network. We pick a number of sites to store the replicated data and to execute transactions. We shall call these the **replication sites**. We make the following assumptions about the network and timing.

1. Each site has its own memory and there is no shared memory.
2. Each replication site has a unique ID.
3. Message size is bounded by *M-size* (bits). Messages sent from site s to s' are received in their sending order. This can easily be achieved by some network protocol.
4. Each replication site has a clock which behaves as a non-decreasing counter in the union of all the time

periods that the site is operational. Also, each clock is precise enough to distinguish the ordering among all instances of the following events: submission of a transaction at the site, execution of a transaction operation, arrival or delivery of a message. The clocks of any two replication sites are synchronized to within a small delta of each other.

5. We assume that sites may suffer from fail-stop failure. This means that when a site fails, it stops processing, and hence a site is either working correctly (is operational) or not working at all (is down). We also assume that communication links are subject to failures. We assume only failures that result in loss of messages. Further more, we assume that each communication link between any 2 sites is a 2-way connection so that if the link between A and B fails, then both communication from A to B and that from B to A are disabled. With this assumption, network partitioning can occur, meaning that the network is divided up into two or more components, where every two sites within a component can communicate with each other, but sites in different components cannot. We assume some mechanism to detect the above failures and to detect their recovery within a reasonable amount of time.

3 Transaction and Data Model

We assume that user jobs are carried out in the form of *transactions*. A transaction is a particular execution of a program that manipulates the data. A transaction may access a data object X by operations $READ(X, y)$ and $WRITE(X, v)$. A $READ(X, y)$ operation reads the value of X and returns it in variable y. $WRITE(X, v)$ changes the value of X to that of v. In addition, each transaction contains a $COMMIT$ or an $ABORT$ as its last operation.

As described in the network model, we select a subset of all sites to be the replication sites. These are the sites that will actually execute the transactions. The remaining sites will interact with both users and some replication site and relay transactions and results between the two. When a transaction is first sent to a replication site, we say that the transaction *originates from* (*is submitted at*) this replication site. We now define *shared-private data* and *public data* in terms of transaction operations. A logical data object X can then be of either one of the following two types:

- **Shared-private data** X **owned by site** s – only transactions submitted at one particular replication site s can perform $WRITE(X, v)$; and these transactions access only shared-private data at s; transactions submitted at other replication sites may perform $READ(X, y)$. We say that s is the *owner site* of X, X is *local* to s and *remote* to the other sites.
- **Public data** X – transactions submitted at any site can perform $WRITE(X, v)$ as well as $READ(X, y)$.

We make the assumption here that all shared-private and public data are fully replicated at the replication sites, i.e. each of them has a copy at every site.

We now identify two main types of transactions:

- **Local Transaction** – a transaction submitted at a site s accessing (i.e. reading and/or writing) only logical data objects that are owned by site s.
- **Public Transaction** – non-local transaction that may read public and/or shared-private data, and that can write only the public data.

We make the following assumptions about transactions:

1. The set of logical data objects that a transaction reads (writes) is called its *readset (writeset)*. In our model, each public transaction pre-declares supersets of its readset and writeset.
2. We assume that the execution of each public transaction is *deterministic*. That is, when a public transaction operation p is executed on its own under a certain state of relevant parts of the database, there is only one possible outcome, namely only one possible return value for p and one possible resulting state of relevant parts of the database. In other words, public transactions do not make purely random choices in its execution.

We could have allowed transactions that read/write public data and write shared-private data. However, these cases can be handled by making such shared-private data public data, and also the consideration of such transactions does not seem to give us advantage in terms of replication control. We clarify that TRS is not trying to monitor the data security among the sites, rather, we are only trying to make use of the site autonomy features to enhance our replication control protocol, data security should really be handled by some other parts of the system.

4 Histories and One-Copy Serializability

In a distributed database system, data objects may be replicated at different sites. The copy of a data object X at site i is denoted by X_i. A data object and its copies are called **logical data object** and **physical data objects**, respectively. The user, when writing transactions, specifies accesses to logical objects. When a transaction T_i executes, the system uses a translation function τ_i to translate **logical operation** into a set of one or more **physical operations**. i.e. $W_i[X]$ is translated into $W_i[X_a]$, $W_i[X_b]$, ..., $W_i[X_l]$, where $X_a, ..., X_l$ are some copies of X and $R_i[X]$ is translated into $R_i[X_1]$, $R_i[X_2]$, ..., $R_i[X_m]$, where $X_1, ..., X_m$ are some copies of X.

The execution of a set of transactions in a distributed database system with replicated data objects can be modeled by a **replicated history** (similar to rd log defined in [BeG81]). A set T of transactions is a partially ordered set $\{T_i = (\Sigma_i, <_i)\}$ where Σ_i is the set of reads and writes issued by transaction i, and $<_i$ tells the order in which those operations execute. A replicated history over such a set T is a partially ordered set $L = (\Sigma(T), <)$ such that

1. T contains two fictitious transactions T_0 and T_f. T_0 is translated into a set of physical write operations, one for each copy of each data object, and these precedes all other physical operations. T_f is translated into a set of physical read operations, one for each copy of each data object, and these are preceded by all other physical operations.
2. $\Sigma(T) = \cup_{i=0}^{f} \tau_i(\Sigma_i)$, where τ_i is the translation function for T_i;
3. for each i and any two operations p_i and q_i in Σ_i, if $a \in \tau_i(p_i), b \in \tau_i(q_i)$ and $p_i <_i q_i$, and if a and b operate at the same site s (possibly on different data X_s, Y_s), then $a < b$;
4. all pairs of conflicting physical operations are $<$ related (two physical operations conflict if they operate on the same physical copy of a data object and at least one of them is a write operation); and

(3) in the above states that we are only interested in replication histories in which the ordering of logical operations within a transaction is preserved by its physical replicas at each site. A committed transaction T_j **reads** X **from** another transaction T_i in a replicated history L = $(\Sigma(T), <)$ if there exists a copy X_a such that

1. $W_i[X_a]$ and $R_j[X_a]$ are operations in $\Sigma(T)$;
2. $W_i[X_a] < R_j\ [X_a]$; and
3. there is no $W_k[X_a]$ such that $W_i[X_a] < W_k[X_a] < R_j\ [X_a]$.

T_j may read X from two or more transactions, each physical read operation being performed at a different copy. We have a one-to-one read-from relation if for each transaction T and for each X that T reads, T reads X from exactly one transaction.

In TRS, public transactions are replicated on multiple sites. We shall call the transactions(operations) intended by users to work on the logical data objects as **logical transactions(operations)**, and the replicated public transactions actually executed at each site accessing physical data copies the **transaction(operation) replicas** of the logical transaction.

A replicated history L_1 is **equivalent** to another history L_2 if both L_1 and L_2 have the same read-from relation. A history H is *serial* if for any two transactions T_i, T_j that appear in H, either all operations of T_i appears before all operations of T_j or vice versa. A 1 copy serial (1C-serial) history is a serial history that consists only of logical operations. An replicated history is **one-copy serializable (1C-serializable)** if it is equivalent to a 1 copy serial history over the same set of logical transactions.

5 Transaction Broadcasting Scheme

In this section, we focus on the basic broadcasting scheme. We shall forget about partition failure for the moment, and assume that each site can communicate with each other site in the network. When we introduce partition failure handling later, the discussion on the set of replication sites here will be restricted to a subset of all these sites (a *view*).

To simplify our discussion, we assume no user-interactions within each public transaction.[1] Public transactions are collected and executed in batches. *Unique transaction timestamps* based on clocks will determine the serialization order among public transactions. We shall ensure that the timestamps of all public transactions follow the order of the transaction submission times measured by the local clocks. Consider a short period of time $[t_1, t_2)$, and consider the batch B of transactions submitted during this period at all sites (each site determines the period by its own clock). B may contain both public and local transactions. We execute the local transactions in B immediately at their origin sites, while the public transactions in B submitted at each site are essentially broadcast and executed on all replication sites. This means that a local transaction and a public transaction replica in B are executed at different times.

Consider for example, two sites s_l and s_g, and suppose that a batch of transactions B submitted in a time period of length δ consists of only two transactions, T_l and T_g, where T_l is a local transaction submitted at s_l and T_g is a public transaction submitted at s_g. T_l is executed and committed immediately during the current period of length δ, while the updates of T_l, and the transaction T_g, are broadcast by s_l and s_g, respectively, at the end of the period and will arrive at the other site later. *Our strategy is to ensure that the result is equieffective to a serial schedule where local*

[1] This limitation can be relaxed (details omitted).

transactions are executed immediately before the public transactions submitted in the same period. Based on this strategy, if T_l writes a shared-private data object X and T_g reads X, then T_g must read the value of X from T_l (since T_l is the only transaction in B that writes X). This implies that s_l must remember the value of X updated by T_l until T_g arrives. This value of X may become an old version in case it is overwritten by another local transaction submitted at a later time at s_l after T_l, and before T_g arrives. In general, our scheme requires each site to remember *old versions* of shared-private data it owns.

The basic step of our protocol for each site s is: *accumulate public transactions submitted at s for a time period of δ and then broadcast the public transactions at the end of the period. If no transaction is accumulated then a null message is sent.*

Definition 1: The set of clock values is a set of real numbers which can be divided into intervals of $[t_1, t_2)$, where $t_2 - t_1$ is a constant value equal to δ, so that the first interval begins at 0, and if the *n-th* interval is $[t_i, t_j)$, then the *n+1-th* interval begins at t_j. Each of these intervals is called a **TRS period**. δ is the length or duration of a TRS period. A TRS period of $[t_1, t_2)$ determines what happens at each operational site when the local clock has value from t_1 to just before t_2, we say that the period **begins at** t_1. A **TRS time** of t_1 determines what happens at each operational site when the local clock has value t_1.

Transactions are collected in TRS periods of $[t_1, t_2)$, and the collected public transactions are **broadcast at** t_2. (This means that at each site s, transactions are collected in $[t_1, t_2)$ and broadcast at t_2 at s's time.) In our discussion we sometimes refer to "TRS period" simply as *period* or *time period.*

Definition 2: A batch of public transaction collected in a TRS period of $[t_1, t_2)$ at a site s is called a **local batch of** s **at** t_2. We define **global batch at time** t_2 as the set of all the public transactions collected at each replication site in period $[t_1, t_2)$.

After every δ time units, s starts the next period of global transaction accumulation and broadcast. For example, if $\delta = 0.5$, then each site broadcasts at times 0.5, 1.0, 1.5, 2.0,..., of its local clock. If we concatenate the submission time of each global transaction with the unique site ID, then we get a globally unique timestamp for each global transaction. The timestamps provide a total ordering on all the global transactions.

Once a site s has received messages from all sites (including itself) broadcast at the same TRS time, it executes the global batch collected based on their timestamp order. A conservative timestamping protocol will be used (see Appendix A).

5.1 Local Transactions

We assume some local concurrency control mechanism to handle the local transactions. The requirement of the local concurrency control is that it generates serializable and *recoverable* [BHG87] histories. A history is recoverable if each transaction commits after the commitment of all transactions (other than itself) from which it read. We shall keep multiple versions of shared-private data for the execution of global transactions. For the execution of local transactions, we need only consider the latest version at any time.

Local transactions can be scheduled for execution immediately upon submission according to some local concurrency control scheme. At the end of each TRS period, s broadcasts the latest committed values of shared-private data updated by local transactions committed in that period, together with the local batch of public transactions accumulated during that period at s. The value of a shared-private data object X that is sent by s in each broadcast is stored as a version of X at s.

Definition 3: We say that for each shared-private data X owned by site s, s **implicitly broadcast** the latest committed version of X at each TRS period that begins at t_i. This version of X is called the **virtual version of** X **at** t_i.

If X is not updated in the period, then the current committed value of X is implicitly broadcast, though in fact no version is physically broadcast.

5.2 Public Transactions

After a global batch has arrived, site s examines the messages that have been received from the other sites, which may contain the new versions of the senders' shared-private data as well as a new global batch of public transactions. The new versions of the shared-private data are first written to the local copies of the shared-private data. Site s then executes the public transactions in the global batch according to the timestamping protocol. We enforce the following rule.

When a public transaction T, which is broadcast by a site s_2 at time t (at s_2's clock) and executed at site s_1 later, reads a shared-private data object X, T should read the virtual version of X at time t, which was implicitly broadcast by some site s_3 at time t (at s_3's clock) where s_3 is the owner of X ($s_3 = s_1$ or $s_3 = s_2$ is possible).

If a physical version actually exists for a virtual version of a shared-private data item at time t, then this physical version is discarded when the execution of the global batch at t is finished. (The term "finished" is defined in Appendix A)

Example: An example is illustrated in Figure 1. In the figure, a horizontal axis represents time measured by site s's clock, and the transmission of messages is indicated by a slanted arrow. The tail of an arrow rests on the time axis at the message sending time t of s, and the head of the same arrow, when projected on the time axis, corresponds to the time (s's clock) at which every message broadcast from every other site s_i at time t (according to s_i's clock) have arrived at s.

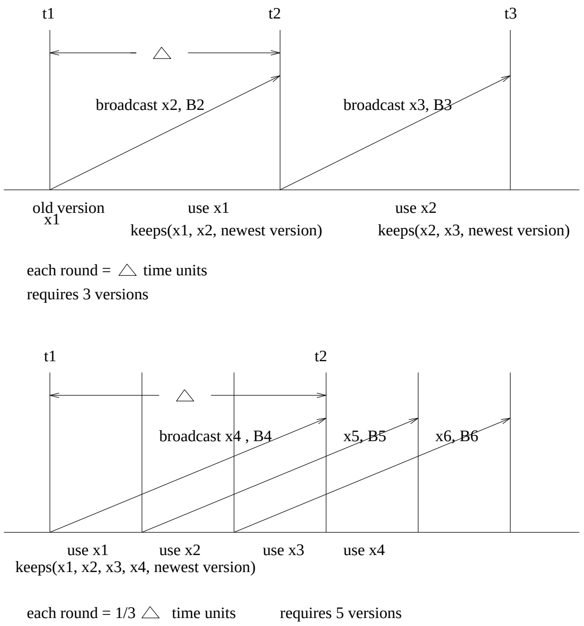

Figure 1: Transaction broadcast

In figure 1, we assume a site s owns a shared-private data object X with consecutive versions, x_1, x_2, x_3, x_4, x_5, x_6. When site s broadcasts a new version x_i (as shown by a slanting arrow), it also broadcasts a new local batch of public transactions B_i submitted at s. Execution can be started once all messages of a period are received at a site. For example, in Figure 1, a site can start executing a global batch of public transactions B_2 at time t_2. In our figures, we assume that all messages transfer takes Δ time as measured by s. In the upper half of the figure, between t_2 and t_3, site s must keep x_2 since execution of B_2 uses x_2, it keeps x_3 because B_3 may need x_3, it also keeps the newest version of X which is used by the current local transactions. The lower half of the figure shows that for TRS periods of length $\Delta/3$, 5 versions of shared-private data may be kept. □

In general, if a message sent at TRS time t at site s arrives at another site s' at TRS time t' where $t' > t$ and $t' = t + \Delta$, if each TRS period has length Δ/q (i.e., $\delta = \Delta/q$), and if the execution of a global batch of transactions can be completed in δ time units, then in the worse case, $\lceil q \rceil + 2$ versions of some local shared-private data may be required at s'. If $t' \leq t$ above, then at most 2 versions are needed.

On the average, the system should be able to handle the execution of a global batch within one TRS period, otherwise the system is receiving more work than it can manage. Given this is true, if the average message delay D is less than δ and if clocks are closely synchronized, then in most cases only a maximum of 3 versions are required. If a shared-private data has not been updated in a period, then obviously no extra version is needed for this period. However, if clocks are not closely synchronized, a message sent at TRS time t may arrive at TRS time t' that is much greater than t, and many versions of local shared-private data may be needed at the receiver's site.

In normal cases, no extra versions of "remote" shared-private data need to be stored; public transactions and the remote shared-private data they may read arrive in the same global batch. Such versions are needed only if clock synchronization is poor or the transaction takes a long time to finish (details omitted).

6 Partition Failures and Commit Protocol

In previous section we have not described the strategy for commit/abort of public transactions, and we did not consider failure conditions. This section describe how to solve these problems with a quorum based protocol and a 2-phase commit protocol.

6.1 Virtual Partition Protocol

For replicated data control under partition failures, quite a number of protocols have been designed. To find one suitable for TRS, we need only observe that the execution of public transactions on public data is quite similar to the "read one write all" special case of the quorum consensus protocol ([Gif79], [ES83]) Although for each logical read, the replicated transaction at each site reads from a different copy (local copy), we ensure that each copy read will be identical. Among the protocols for partition failure, the virtual partition protocol ([ET86], ET89]) is one that can maintain the "read one write all" characteristics where "write all" means writing to all copies in a "virtual partition".

A generalized version of virtual partition protocol, GVP, is given in [Fu90]. Here we give a brief outline of GVP. Each transaction is executed under a **view** which is a subset of the set of all replication sites. (The view of a site is what the site considers as the partition it currently belongs to, it is the "virtual partition" seen by this site.) We summarize the major criterion of the protocol here.

1. **View-id** : Each user transaction **executes in a view**. Each view has a unique view-id, *V_id*. Trans-

actions executing in a view are controlled by a concurrency control protocol within the view. Each copy of a data object has a **version number** = $< V\_id, k >$, indicating that it was last written in view V with view-id V_id, and that its value is the result of the kth update in that view. (Uniqueness of view-id can be enforced by concatenating a sequence number with the site ID.)

2. **Global read quorums** : For each data object X, a global read quorum set $RQ(X)$ is defined. X is **inheritable** in view V if V contains a global read quorum belonging to $RQ(X)$.

3. **View quorums** : For each data object X, and each view V, a **view read quorum** set $rq(X, V)$ and a **view write quorum** set $wq(X, V)$ are defined. Each view write quorum in $wq(X, V)$, if any, intersects each quorum in $RQ(X)$ and each view read quorum in $rq(X, V)$, if any. If $wq(X, V) \neq \phi$ ($rq(X, V) \neq \phi$), then X is said to be **writable (readable)** in V.

4. **view updating** : When changing its view to a new view V', V' must have a view-id greater than the old view, a view update transaction in view V' atomically **initializes** copies of each data object X inheritable in V' by reading the most up-to-date copy of X in a global read quorum in $RQ(X)$, where each accessed copy must have a version number $< V\_id, k >$ with $V\_id \leq V'\_id$.

5. **Reading and writing**

   A logical write of X by a user transaction T executing in view V with view-id V_id is required to update/**initialize** all copies in a view write quorum in $wq(X, V)$ (where each copy has a version number that contains a view-id $\leq$ V_id), giving them the same new version number $< V\_id, k' >$ larger than their previous version numbers.

   A logical read of X by user transaction T is allowed in view V only if at least one copy of X accessed by the read has been *initialized* in V. A logical read of X by T reads a view read quorum from sites that have the same view as T and take the value of the copy with the highest version number.

For TRS, GVP is adopted as follows: for a public data X, if a view V is $\{ s_1, s_2, ..., s_n \}$, and if V contains a quorum in $RQ(X)$, then $rq(X, V)$ is the set $\{ \{s_1\}, \{s_2\}, ... \{s_n\} \}$ (in fact, a site will read from its local copy, but we shall show that each replicated read operation reads the same value), and $wq(X, V)$ is the set $\{ \{s_1, s_2, ..., s_n\} \}$. If V does not contain any quorum in RQ(X), then both $rq(X, V)$ and $wq(X, V)$ are ϕ. From this definition, we can deduce that any two quorums in $RQ(X)$ must have a non-empty intersection. This is because any two different views V_1, V_2 each containing exactly one quorum in $RQ(X)$ have view write quorums equal to V_1 and V_2, respectively. And V_1 must intersect V_2 since a quorum in $RQ(X)$ must intersect a view write quorum in any view. Examples of $RQ(X)$ are primary site and majority coteries.

Since a shared-private data X can be updated only by its owner site s, we make $RQ(X)$ equal to $\{\{ s \}\}$. $rq(X, V)$ and $wq(X, V)$ are also $\{\{ s \}\}$ for any view V if s is in V, otherwise, $rq(X, V)$ and $wq(X, V)$ each equals the empty set.

6.2 Decentralized Two-Phase Commit

A commit protocol is necessary to ensure that whenever one site decides to commit(abort) a public transaction, every other site must also decide to commit(abort) the public transaction.[2]

In TRS, we have two types of "commit/abort". The first type is the conventional commit/abort of individual transactions. The second type is the "commit/abort" of global transaction batches, where the commit/abort of individual transactions have been tentatively decided by each site. If the global batch commits, the tentative commits/abort of transactions in the batch are realized. If the global batch aborts, then all transactions in the batch abort. Public transactions batches are committed in a two-phase consensus. We essentially adopt a decentralized two-phase commit protocol [Ske82] for each global batch. This is illustrated in figure 2.

- **[Phase 1]:** Transactions are broadcast from all sites to all sites in a view V. When a site s has view V, it expects transaction batches from all sites in V at each TRS period. On receiving such a global batch, it will execute the transactions in the batch. It will broadcast the tentative decision of the site on the commit/abort for each transaction in the batch. For example if there are 3 ordered transactions T_1, T_2, T_3 in a batch, and s tentatively decides to commit T_1, T_2 but abort T_3, then its decision will be { commit, commit, abort }. We define a special $\perp$ **decision**. A $\perp$ decision is not **identical** with any other decision, a $\perp$ decision is not identical with another $\perp$ decision. There are a number of cases where a site would broadcast a $\perp$ decision:

[2] For a commit protocol for a transaction that is executed at more than one site in a distributed systems, Skeen [Ske82] proved that if a network partition failure occurs during the execution of such a protocol, then no atomic commit protocol can guarantee the termination of the transaction involved (as long as the partition persists), i.e., partitions may prevent some transactions from terminating. In other words, there are no **non-blocking** protocols for partitionings.

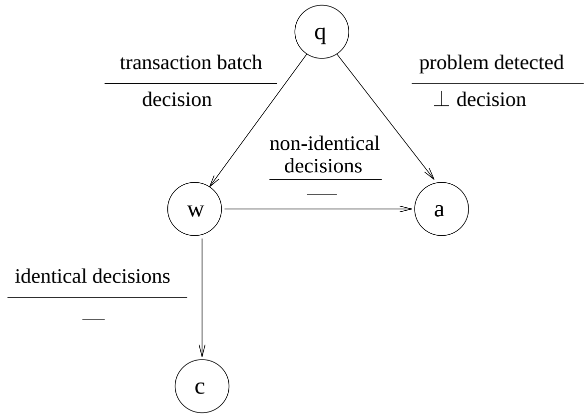

Figure 2: Two Phase Commit Protocol

1. If site s has view V and it receives some message (either transaction batch or view-update transaction) from a site s' with a view of V', where $V' < V$, then s will response to s' with a $\perp$ decision message, and also V may be sent along, so that s' may update itself to V.
2. If s has view V and it receives some message from another site s' with view V', where $V' > V$, then s may decide to join the view V' and generate a $\perp$ decision to transactions of view V which are at state (q).
3. s may have detected some network failure or recovery, and decide to change its view. Hence it generates a $\perp$ decision to transactions in the current view V. (This includes the case where an expected transaction batch is not totally received because of some failure.)

If a site sends out a $\perp$ decision, it can go to state (a) since the decisions are guaranteed not to be identical.

- **[Phase 2]:**

  If a site receives identical decisions from all sites in the view, it can commit the batch. The transaction batch moves from state (w) to state (c).

  s decides to abort when it receives non-identical decisions from two or more sites in V. The transaction batch moves from state (w) to state (a).

If s does not receive an expected message from some site in state (w) then a *termination protocol* (see below) is carried out. It is possible that the batch of transactions are *blocked* by the termination protocol.

In our timestamping protocol (Appendix A) we allow transactions to read from transactions that have not committed or aborted, hence when we decide to abort a global batch, subsequent batches in the same view may also need to be aborted. However, since we have made all user transactions replicas to execute under essentially the same database state at all sites under normal circumstances, most execution decisions will be unanimous. Transaction batch aborts are triggered only by exceptional cases, such as problems at some site, site failures or partition failures.

6.3 View Update in TRS

A site can initiate a view-update on detecting failure or recovery. It may also adopt a new view that it becomes aware of. When a site updates its view, transactions executing in older views that are not ready to commit will be aborted.

In the process of view-update, we try to update the views of all the sites in a new view, and we want to bring up-to-date all copies of all public data items for which there is a global read quorum in the new view. View update is executed as a special transaction issued by the system and not by the users. The view-update transaction is initiated by s by broadcasting the new view-id to all sites in V. During view update, we make sure that for each public data object X, if a quorum in $RQ(X)$ exists in the new view V (data X is inheritable), then each site from a quorum in $RQ(X)$ broadcast its version of X to all sites in the new view V. Whether s is initiating or adopting a view, s waits for enough messages which contains values of X at different sites in view V. When enough copies in a quorum in $RQ(X)$ is received, s can update(initialize) its copy of X to be the value of the copy with the greatest version number.

View-update transaction also follows a 2-phase commit protocol as follows: In the first phase, sites receives notification of the view-update and each either disagrees to join by broadcasting a negative acknowledgement or agrees to join and broadcast their versions of public data as needed. In the second phase, each site determines if all sites in V agree to join the view, in which case the view is updated and it updates public data replicas with the latest version if possible; otherwise, the view-update is aborted.

We make sure that view update is successful only if all sites in the new view V agree on the view V. This is because each site must later participate in the transaction broadcasting scheme. After committing the view-update transaction successfully, all sites agrees on the next TRS period to begin transaction broadcasting in the new view.

For shared-private data X, $RQ(X)$, $wq(X,V)$ and $rq(X,V)$ each equals $\{\{s\}\}$, where s is the owner site of X. Hence view update does not update copies of X in the new view. At the first TRS transaction broadcasting of a new view, the latest committed version of each shared-private data is broadcast at the same time. This allow all replication sites to bring up-to-date its local copies.

6.4 Termination Protocol and Recovery

The termination protocol and recovery mechanism will follow the conventional ideas for such schemes. The execution of a transaction can be blocked at a site in the termination protocol. Note that while a blocking exists in view V so that some data X are suspended, a view update from V to V' initiated in this partition will not be able to read from a quorum in RQ(X). However, some other view update from another view V'' ($\neq V$) to V''' ($> V$) may be able to update X. Hence we allow writes to some data although a previous transaction which tried to read/write the data has not committed or aborted. However since those operations are from an older view, they can be ignored in the new view.

7 Correctness

In section 4, we defined a replicated history over a set T of user transaction, which includes T_0 and T_f, to be a partially ordered set $L = (\Sigma(T), <)$ such that $\Sigma(T) = \cup_{i=0}^{f} \tau_i(\Sigma_i)$, where τ_i is the translation function for T_i. For a history generated by TRS we must extend this definition since we have introduced some mechanisms on top of the user transactions which may affect the database state. There are two types of such mechanisms in the system: (1) view-update transactions and (2) broadcasting of versions of shared-private data, and the updating of local copies of each site with such versions.

Let us introduce *virtual transactions* to represent the second type of mechanism. For each broadcasting and the corresponding update of a shared-private data X at remote sites, we create a virtual transaction whose translation reads the value of the replica of X at the owner site s immediately after the value is committed, and writes the value of X at the other sites at the corresponding time they are updated. In order to assimilate these mechanisms, we modify the definition of a **replicated history** so that $\Sigma(T)$ also contain the translation of the view-update transactions and the virtual transactions.

A **TRS history** (history generated by TRS) is a replicated history generated by TRS over some set of user transactions, where if a $READ/WRITE$ logical operation of a user transaction is translated into a number of physical operations, then each of these physical operations belongs to one transaction replica.

Next we need to augment the definition of the read-from relation. This is because we have incorporated some mechanism to pass information from one transaction to another, other than direct transfer by the orderings of write and read operations of user transactions on the same physical data object. A sequence of view-update transactions $VT_1, ...VT_n$ (n may be 1) may convey the value v_1 of some data from one copy c_1 to other copies C_1, let T_p be the user transaction that has written this value in copy c_1. If a transaction T reads the value v at some site in C_1 written by VT_i for some i, $1 \leq i \leq n$, we say that T **reads** the value **from** T_p. Also there is the broadcasting of new value v_2 of shared-private data from the owner site copy c_2 to update other copies C_2 (by virtual transactions above). let T_l be the user transaction that has written this value at c_2. If a transaction T reads the value of v_2 at some copy in C_2, we say that T **reads** the value **from** T_l. Hence we extend the meaning of read-from with the above augmentations.

An replicated history L_1 over a set of user transaction is **equivalent** to another history L_2 over the same set of user transactions if both L_1 and L_2 have the same read-from relation among the user transactions and T_0, T_f. If a TRS history α over a set of user transactions is equivalent to a 1 copy serial history over the same set of user transactions, then α is said to be **1C-serializable**.

In order to show that a TRS history α is 1C-serializable, we need to find a 1C-serial history β that contains all the logical public transactions corresponding to the transaction replicas in α and to show that the following conditions hold.

1. A replica T_r of logical public transaction T reads from T'_r (a replica of the logical public transaction T') in α iff T reads from T' in β, and T_r reads from local transaction T'' in α iff T reads from T'' in β.
2. The read-from relation among local transactions are the same in α and β.

Since correctness of the virtual partition protocol and the decentralized two-phase commit protocol has previously been established, we only focus on the features that are specific to TRS. In the following, for a replication history α, we call the sub-history of α that contains all operations of the committed transactions in α a *committed history.* We say that a set of transactions can be *serialized* (are serializable) in some order if there exists an equivalent serial history on the same set of transaction that has the same ordering.

Theorem 1: Each committed history generated by GVP is a 1-Copy serializable histories.

Proof sketch: We compare GVP with the virtual partition protocol. GVP differs from virtual partition protocol in 2 ways. The first difference is that virtual partition ensures quorum intersection properties by means of the number of copies in a quorum, while GVP has enforced the intersection properties explicitly in terms of set intersections. The second difference is that GVP allows a data item X to be initialized by a write operation of a view V if the view has a quorum in $wq(X, V)$. Hence even if X is not inheritable in V, it has a chance to be initialized and become accessible. First we observe that when a quorum in $wq(X, V)$ in V exists, no other view V' that is disjoint from V can inherit X. This is because any quorum in $RQ(X)$ has to intersect any quorum in $wq(X, V)$ that exists in V. Hence the only possibility that transactions in concurrent views can operate on X is that X is also initialized by a write operation in another view V'. Since the outcome of the initial write operation does not depend on the operations on X in the other views, we can still serialize the transactions in the order of the view-ids as in virtual partition protocol. □

Lemma 1: A site eventually commits/aborts a transaction replica if and only if every other site eventually commits/aborts its transaction replica of the same logical transaction.

Lemma 1 is enforced by the decentralized 2-phase commit protocol. □

With Lemma 1 we can talk about a committed public transaction in a TRS history, which is a public transaction with each transaction replica committed at its replication site.

Lemma 2: In a TRS history, the values read/written by each read operation replica of a logical read/write operation of any committed public transaction are the same.

Proof sketch: The view update transaction ensures that all copies inheritable in a view are the same before any transaction is executed in the view. A global batch of transactions read from the virtual versions of shared-private data that are implicitly broadcast at the same time, hence the values of such data they read are the same. If a transaction batch commits, since transaction execution is assumed to be deterministic, the timestamping protocol makes each of the global transaction batch replicas in the execution's view equivalent to a one-copy serial history with the transactions executed in the unique timestamp order. □

Theorem 2: In our transaction model and network model, each committed history generated by the TRS scheme is one-copy serializable.

Proof sketch: We can show that a committed TRS history is equieffective to a history generated by GVP on the same set of transactions.

TRS differs from GVP since there are transaction replicas that read and write the data copies at each site. From Lemma 2 above, we see that each operation replica of a logical operation will read/write the same value. It follows that the public transactions in a view V broadcast at each TRS period can be seen as a sequence of transactions that follows GVP by using a read one write all quorum consensus.

We can show that for any committed TRS history α, there is an equivalent 1C-serial history β in which

1. transactions of different views are ordered by their view-id, so that all transactions with a view V are scheduled after all transactions with a smaller view and before all transactions with greater views;
2. within a view, public transactions are ordered by their timestamps, and
3. within a view, the local transactions (LT) that commit in any period of $[t_i, t_i + \delta)$ in α appear after all committed public transactions in α broadcast at time t_i and before all committed public transactions in α broadcast after time t_i.

(1) is from the correctness of the virtual partition protocol and GVP. (2) is from the conservative timestamping protocol. Proof of (3) is omitted. □

8 Performance

On the average, under non-faulty situation, for a batch of transactions, the time that elapses from the time of submission of a transaction T to the time of commit of T is $2D + \frac{\delta}{2} + \tau + \alpha$, where D is the average communication time, δ is the TRS period, τ is the average time to execute a batch of transactions, and α is the average real time deviation between 2 clocks in the system (the real time needed for one clock to catch up with another clock). $2D$ is the time to send a batch of transaction, plus the time to send the resulting decision responses. $\frac{\delta}{2}$ is the average time a transaction waits before the sending of the transaction batch. If user interactions within a transaction is necessary, then more communication delay is inevitable (details omitted).

Let m be the number of replication sites. For a batch of public transactions, message broadcasting in TRS uses $2m^2$ messages. However, m^2 of these messages are short messages containing only the commit/abort decisions on transactions and they can also be piggybacked with other broadcasting messages.

If the response messages are not sent immediately after execution, but wait until the next transaction broadcasting to be sent along, then the average response time will be $2D + \delta + \tau + \alpha$.

Conflicts exist among transaction operations if they try to access the same data item and one of them is a write operation. In TRS, if transaction T_1 conflicts with transaction T_2 and T_1 has a smaller timestamp, then if T_1 does not commit for some time, T_2 also cannot commit. We believe that TRS behaves well in conflict cases since such "locks" on data are not held over many communication delays.

9 Conclusion

Liskov in [Lis91] has advocated the use of clock synchronization which is quickly becoming a reality in wide area network. To our knowledge, TRS is the first distributed database scheme proposed to make use of clock synchronization and transaction replication to achieve better performance in concurrent transaction execution. With TRS, execution of a public transaction normally incurs only 2 communication delays. Hence TRS is more efficient than centralized transaction management schemes in terms of communication delay when we have transactions with multiple interdependent operations. TRS does not normally lock data while waiting for a number of communications delays, and by a conservative timestamping scheme, TRS will not trigger any transaction abort due to conflicts among concurrent transactions. This will be significant for "hot-spot" data where conflicts are frequent. If shared-private data type exists, TRS makes use of the semantics of node autonomy to allow local transactions to be executed under local concurrency control.

If public transactions are not frequent then TRS may generate a lot of wasteful null messages. Hence we may consider TRS more useful for busy systems. However, in real applications, we may have dedicated data channels which will be allocated even if no messages are sent. In that case, null messages do not cost more network resources. TRS transmits transactions instead of data items, hence if the size of transactions is considerably greater than the size of data accessed, then the communication overhead of TRS in terms of bandwidth is greater; otherwise, TRS is better off. TRS repeats the execution of each transaction at multiple sites, hence it incurs more computation overhead if the transactions require a lot of computation, but we believe that a lot of business applications are more I/O-oriented than computation-oriented. We require storage for keeping multiple versions of shared-private data, if this becomes a problem, we can make all data public.

As discussed in [PiG89], replicated execution enables the detection of computational errors, we may modify the commit protocol so that each site compares the outcome of all sites in terms of computational results of the operations, and may correct errors found. When this checking is enforced, it is possible to have two or more different software versions for the same public transaction, which helps us to discover vicious coding errors. We may also consider using a three-phase commit protocol and derive some better termination protocol ([Ske82], [CK85], [RL92]). Finally we may need more elaborate performance analysis.

Acknowledgements

We would like to thank Prof. Tiko Kameda for a major improvement on the generalized virtual partition protocol, and other helpful suggestions.

References

[BeG81] P.A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys* 13(2):185-221, June 1981.

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

[CK85] D. Cheung and T. Kameda, Site Optimal Termination Protocols for a Distributed DB Under Network Partition, *ACM 4th Symposium on Principles of Distributed Computing*, 1985.

[CS80] D. Clark and L Svobodova, Design of distributed Systems supporting Local Autonomy, *Proc COMPCON-IEEE*, p.438-444, Spring 1980.

[ES83] D.L. Eager, K.C. Sevcik, Achieving Robustness in Distributed Database Systems, *ACM Transactions on Database Systems* 9(4):560-595, Dec 1984.

[ET86] A. El Abbadi and S. Toueg, Availability in Partitioned Replicated Databases, *ACM 5th Symposium on Principles of Database Systems*, 240-251, March, 1986.

[ET89] A. El-Abbadi and S. Toueg, Maintaining Availability in Partitioned Replicated Databases, *ACM Transactions on Database Systems* 14(2):264-290, June 1989.

[Fu90] A.W. Fu, Enhancing Concurrency and Availability for Database Systems, *Ph.D. Thesis, Simon Fraser University* April 1990.

[GaK88] H. Garcia-Molina and B. Kogan, Node Autonomy in Distributed System, *Proc Int'l.Symp. on Database in Parallel and Distributed Systems,* p.158-166, Austin, 1988.

[Gif79] D.K. Gifford, Weighted Voting for Replicated Data, *Proc 7th ACM SIGOPS Symposium on Operating Systems Principles*, 150-159, Dec 1979.

[KoG87] B. Kogan and H. Garcia-Molina, Update Propagation in Bakunin Data Networks, *ACM 6th Symposium on Principles of Distributed Computing*, p.13-26, 1987.

[Lam78] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed Multiprocess System, *Communications of the ACM* 21(7):558-565, July 1978.

[Lis91] B. Liskov, Practical Uses of Synchronized Clocks in Distributed Systems, *ACM 10th Symposium on Principles of Distributed Computing,* p.1-8, 1991.

[LS80] B. Lindsay and P.G. Selinger, Site Autonomy Issues in R*: a Distributed Database *Management System, Research Report, IBM Research Division,* 1980.

[Mil88] D.L. Mills, Network Time Protocol (Version 1) Specification and Implementation DARPA-Internet Report RFC-1059. July 1988.

[PiG89] F. Pittelli and H. Garcia-Molina, Reliable Scheduling in a TMR Database System, *ACM Transactions on Computer Systems* 7(1):25-60, Feb 1989.

[RL92] M. Rabinovich and E. Lazowska, A Fault-Tolerant Commit Protocol for Replicated Databases, *ACM 11th Symposium on Principles of Database Systems*, p. 139-148, 1992

[Ske82] D. Skeen, NonBlocking Commit Protocols, *ACM SIGMOD Conference on Management of Data*, p.133-147, 1982.

[TT91] P. Triantafillou and D. Taylor, Using Multiple Replica Classes to Improve Performance in Distributed Systems, *IEEE 11th International Conference on Distributed Computing Systems*, May 1991.

Appendix A: Conservative Timestamping

Concurrency control using conservative timestamping ordering [BeG81] does not require transaction abortion. We see that the periodical broadcasting of transactions makes this approach easier because little waiting (depending on δ, the length of a TRS period) is necessary for a site to make sure that no transactions with older timestamps will be received from other sites. We make use of the assumption that each transaction T pre-declares its readset and writeset, denoted by $readset[T]$ and $writeset[T]$, respectively.

Our approach is to preprocess all the transactions in each global batch B in a view to detect read/write and write/write conflicts. Let $T_1, T_2, ...T_n$ be the transactions in B in the timestamp order.

We propose an algorithm that maintains two sets: PRECEDE[T_i, X] and INFORM[T_i, X]. PRECEDE[T_i, X] contains all transactions that accesses X and which should be executed before T_i. Initially these sets are empty.

We shall be careful about commit and abort. A transaction replica can be committed or aborted **tentatively** at its execution site, however, it will later be subjected to a 2-phase commit protocol which decides on the "commit" or "abort" of an entire global batch. A transaction replica commits if and only if it tentatively commits and the global batch it belongs to also commits. If a transaction replica is tentatively aborted, then all its previous operations are undone. Here we say that a transaction replica **finishes** when all its operations (including tentative commit/abort) other than commit or abort have been executed. We assume that transactions in all previous global batches has finished, and execution follows an update-in-place [BHG87] approach.

For each data object X, preprocessing examines each transaction T_i that reads or writes X. If T_i writes X, then the algorithm looks for the latest preceding transaction (in terms of timestamps) T_j that writes X and puts it in PRECEDE[T_i, X]. All the transactions with timestamp between those of T_j and T_i that read X are also placed in PRECEDE[T_i, X]. For each transaction T_j in PRECEDE[T_i, X], T_i is inserted into the set INFORM[T_j, X], so that T_j can inform T_i about the completion of T_j when it finishes. If X is only read by T_i, then the algorithm looks for the closest preceding transaction (in terms of timestamps) T_j that writes X. T_j is then placed in PRECEDE[T_i, X], and T_i is inserted into the set INFORM[T_j, X]. The transaction manager (TM) carries out these operations locally at one site. When T_j finishes, TM erases it from PRECEDE[T_i,X]. T_i cannot access data object X unless PRECEDE[T_i, X] $= \phi$. After the preprocessing, we can start execution.