The Group Paradigm for Concurrency Control Protocols

(Extended Abstract)

Amr El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106

Sam Toueg
Department of Computer Science
Cornell University
Ithaca, NY 14853

ABSTRACT

We propose a paradigm for developing, describing and proving the correctness of concurrency control protocols for replicated databases in the presence of failures or communication restrictions. Our approach is to hierarchically divide the problem of achieving one-copy serializability by introducing the notion of a "group" that is a higher level of abstraction than transactions. Instead of dealing with the overall problem of serializing all transactions, our paradigm divides the problem into two simpler ones: (1) A *local policy* for each group that ensures a total order of all transactions in that group. (2) A *global policy* that ensures a correct serialization of all groups. We use the paradigm to demonstrate the similarities between several concurrency control protocols by comparing the way they achieve correctness.

1. Introduction

The availability of data in a distributed database can be increased by replication. However, the design of distributed databases is often complicated by the need to overcome the unpredictable failure of components. Sites may fail by crashing, or by failing to send or receive messages. Links may fail by crashing, delaying or failing to deliver messages. Site or link failures may partition the database. Sites in a *partition* can communicate with each other, but may not communicate with sites in other partitions [Davidson85]. Communication between sites may also be restricted if communication costs are expensive. The system designer may restrict the execution of some operations to a set of sites, and later share information with the rest of the network.

In this paper we present a paradigm for developing, describing and proving the correctness of concurrency control protocols for replicated databases in the presence of failures or communication restrictions. The correctness criteria considered is *one-copy serializability* [Bernstein87]. The usual approach for achieving one-copy serializability is to use a *concurrency control protocol* [Eswaran76] for synchronizing the execution of transactions, where a transaction is assumed to transform the database from one consistent state to another.

Our approach is to hierarchically divide the problem of achieving one-copy serializability by introducing the notion of a "group" that is a higher level of abstraction than transactions. A *group* is a set of transactions. Instead of dealing with the overall problem of serializing all transactions, our paradigm divides the problem into two simpler ones. Informally, one must first derive a *local policy* for each group that ensures a total order of all transactions in that group. This guarantees that each group, as a whole, transforms the database from one consistent state to another, hence, each group can be viewed as a high level transaction. Then, one should develop a *global policy* that ensures a correct serialization of all groups. The serialization order of all groups, combined with the serialization of all transactions in a group, ensures one-copy serializability.

To prove that our paradigm ensures one-copy serializability, we introduce a model, the *one-copy serialization group graph* that is particularly suited to our concept of groups as a high level transaction. The nodes of this graph are groups (i.e., sets of transactions), whereas in previous methods like the serialization graphs [Bernstein87], nodes are transactions. Using groups instead of transactions as the basic unit simplifies the proofs, and highlights the properties required

to ensure one-copy serializability. We prove that if transactions can be partitioned into groups such that all transactions in each group are one-copy serializable, and there is a corresponding acyclic one-copy serialization group graph, then the execution is one-copy serializable.

Another contribution of this paper is that we describe and prove the correctness of several concurrency control protocols [Eager83, Herlihy87, El Abbadi86, Skeen84] in terms of one model. We show that even though these protocols may appear to be unrelated, they are all instances of our group paradigm. This emphasizes the similarities between them, and hence simplifies their understanding.

In the next section, we briefly describe the standard replicated database model and its correctness criteria. In Section 3 we present the group paradigm and in Section 4 we develop a correctness model that is particularly suitable for the group paradigm. In Section 5 an analysis of several concurrency control protocols is made that emphasizes the fact that they are instances of the group paradigm.

2. The Standard Model

We consider a *distributed database* consisting of a set of *objects*. A *transaction* is a set of read and write operations that are executed as an indivisible unit, and in a certain order on the objects. A transaction is assumed to be correct, i.e., when executed alone on a consistent database it leaves the database in another consistent state. Furthermore, a transaction's execution is *atomic*; i.e., before a transaction terminates it either *commits* or *aborts* all changes that it made to the database.

An object x is implemented by a set of *copies* x_s, x_p, $\cdots$. An operation issued by a transaction on an object is called a *logical operation*. Such an operation is executed by a set of *physical operations* on some of the copies of the object. A logical write operation $w_i[x]$ is executed by a set of physical write operations $\{w_i[x_p], \ldots, w_i[x_s]\}$. A logical read operation $r_i[x]$ is executed by a set of physical *access* operations $\{a_i[x_p], \ldots, a_i[x_j], \ldots, a_i[x_q]\}$ that results in a single value x_j being read by read operation $r_i[x_j]$.

We assume a database model similar to that defined by Bernstein, Hadzilacos and Goodman [Bernstein87], where the execution of operations in the system is modeled by *logs*. A *replicated data log* (or simply a log) L over a set of transactions $T = \{t_1, t_2, \ldots, t_n\}$ is a partial order $(S, <_L)$, where S is the set of all physical operations executed by committed transactions in T and $<_L$ reflects the order in which the physical operations were executed at each copy. A *serial* log is a totally ordered log such that for every pair of transactions t_i and t_j, either all physical operations executed by t_i precede all physical operations executed by t_j or vice versa. A *one-copy* log is a log where each object is implemented by one copy. Transaction t_j *reads_x_from* t_i in log L if there is a copy x_p of object x such that t_i executes $w_i[x_p]$, t_j executes $r_j[x_p]$, and:

1. $w_i[x_p]$ and $r_j[x_p]$ are in L
2. $w_i[x_p] <_L r_j[x_p]$
3. There is no $w_k[x_p]$ such that $w_i[x_p] <_L w_k[x_p] <_L r_j[x_p]$.

We define a write operation $w_i[x_p]$ to be a *final write* for copy x_p in a replicated data log L if for all $w_j[x_p]$ in L $(i \neq j)$, $w_j[x_p]$ precedes $w_i[x_p]$ (in a one-copy log, final writes are defined on objects, since each object is implemented by one copy only). A replicated data log L_1 is *equivalent* to a one-copy log L_2 if (1) for all t_i, t_j, and x, t_j *reads_x_from* t_i in L_1 if and only if t_j *reads_x_from* t_i in L_2, and (2) for each final write $w_i[x]$ in L_2, all write operations $w_i[x_p]$ by t_i into some copy x_p is a final writes on x_p in L_1. A log is *one-copy serializable* if it is equivalent to a serial one-copy log over the same set of transactions.

3. The Paradigm

In this section, we first present the notion of a "group" as a high level transaction, and develop the necessary formal framework for the presentation of the group paradigm. The group paradigm for concurrency control protocols is then presented in two components: the local policies, which ensure the serialization of all transactions in each group; and the global policy, which ensures the serialization of all groups in such a way that all transaction executions are one-copy serializable.

We should note that although our concept of groups might superficially be similar to the *nested transaction* structure [Moss82], there are significant differences. In the nested transaction model, a transaction is designed in a structured manner as a single entity, where a transaction may invoke several sub-transactions, which may themselves invoke other sub-transactions. The nested transaction as a whole is synchronized with other nested transactions at the root level, and the different sub-transactions in the same nested transaction synchronize with one another. The same concurrency control mechanism is used by all transactions, and a special mechanism is used to correctly synchronize the accessing of data by different sub-transactions. In contrast, the group concept is a convenient model for

characterizing different concurrency control protocols, where each group could have a *different* local policy for synchronizing the execution of transactions. Furthermore, the group model does not have any *a priori* assigned structure, and in particular, there is no predetermined precedence ordering imposed on the execution of transactions.

3.1. Groups

Recall that a transaction is composed of a set of operations, which if executed alone, takes the database from one consistent state to another. We now propose a higher level of abstraction than the transaction: a *group*, which is a set of transactions. Formally, given a set of transactions T, a set of groups $\{g_1, g_2, \ldots, g_k\}$ partitions T into disjoint sets, such that $\forall i,j\ g_i \cap g_j = \emptyset$ and $\bigcup_i g_i = T$. The execution of (all transactions in) a group should transform, as a whole, the database from one consistent state to another. It will be useful for the reader to think of groups as higher level transactions.

We now extend the definition of one-copy serialization to the set of transactions in a group (as opposed to the set of all transactions). In the standard model, it is assumed that a single transaction t_{init} initializes the database. Note that transactions in a group may read values written by transactions belonging to several other groups. For a log L ($L=(S_L, <_L)$) and group g, we introduce transaction t^g_{init} that initializes the database from g's point of view. Transaction t^g_{init} contains all write operations executed by transactions not in g, but that write values read by transactions in g, i.e., $t^g_{init} = (S^g_{init}, <^g_{init})$, where $S^g_{init} = \{ w_i[x_p] \mid$ where $w_i[x_p]$ is issued by $t_i \notin g$, and $\exists\ t_j \in g$ such that t_j *reads_x_from* $t_i\}$, and $<^g_{init} = \{ <op_i, op_j> \mid op_i, op_j \in S^g_{init}$ and $op_i <_L op_j\}$.

The sublog L_g is defined as a restriction of L to group g. Sublog L_g is a partial order $(S_g, <_{L_g})$, where S_g is the set of all operations executed by transactions in g, as well as the operations of t^g_{init}, and $<_{L_g}$ is the subset of $<_L$ corresponding to operations in S_g. Formally, $S_g = \{op_i \mid$ where op_i is issued by $t_i \in g$ or $op_i \in S^g_{init}\}$ and $<_{L_g} = \{<op_i, op_j> \mid op_i, op_j \in S_g$ and $op_i <_L op_j\}$.

A group g is *one-copy serializable* if L_g is equivalent to a serial one-copy log with t^g_{init} preceding all other transactions. A serial one-copy log defines a total order $<_g$ over all the transactions in g as well as transaction t^g_{init}. The *final value* of an object x with respect to a group g and total order $<_g$ is defined as the value of x written by the last transaction (with respect to $<_g$) that writes x in g, i.e., the value of x written by the final write operation on x in the equivalent serial one-copy log.

A group g is said to *write* (*read*) an object x, if there is a transaction in g that writes (reads) x. A group g_j *reads_x_from* g_i if g_j reads a value of x written by g_i. Two groups g_i and g_j are said to *conflict* if there are two logically conflicting transactions $t_i \in g_i$ and $t_j \in g_j$.

3.2. Group Concurrency Control Protocols

We now describe a family of concurrency control protocols based on the concept of groups as "high level" transactions. In Section 4 we show that any member of this family ensures one-copy serializability. In Section 5, we show that several previously known protocols are members of this set. A *group concurrency control protocol* π partitions transactions into disjoint groups, and has the following two components:

1. *A Local Policy* P_g, for each group g, ensures the one-copy serializability of (all transactions in) g. The *local order* $<_g$ is defined to be the serialization order of all transactions in g given by P_g.
2. *A Global Policy* P ensures a total order $<$ on all groups such that:

if g_j *reads_x_from* g_i, and $i \neq j$, then

a. $g_i < g_j$.

b. There is no g_k that writes x such that $g_i < g_k < g_j$.

c. g_j reads the final value of x in g_i, with respect to $<_{g_i}$, given by P_{g_i}.

The group ordering $<$ and the serialization ordering of transactions in each group, define a total order O of all transactions. The policies must ensure that this total order is a correct serialization order, i.e., is equivalent to a serial one-copy log. Informally, condition 2.a ensures that the order on groups is consistent with the *reads_x_from* relation. Conditions 2.b and 2.c ensure that in the total order O no transaction t_k that writes x is ordered between two transactions t_i and t_j where t_j *reads_x_from* t_i. In the next section we present a model for proving that O is a correct one-copy serialization order, i.e., group concurrency control protocols ensure one-copy serializability.

4. Serialization Group Model

In this section we show that any group concurrency control protocol ensures one-copy serializability. The local policies ensure that each group g is one-copy serializable with some order $<_g$. We now show that the conditions on the global policy are sufficient to ensure one-copy

serializability of all transactions.

Let L be a log over a set of transactions T that are partitioned into groups. The local policy P_g of group g ensures that g is one-copy serializable according to a total order $<_g$. A graph G is a *one-copy serialization group graph* $1{-}SGG[L]$ for log L over T with a local order $<_g$ for each group g if G satisfies the following conditions:

1. The nodes of G represent the groups that partition T.

2. If g_j *reads_x_from* g_i, then G contains an edge from g_i to g_j. This edge is called a *group reads_from* edge. If the value read by g_j is not the final value of x with respect to g_i and local order $<_{g_i}$, then G must contain an edge from g_j to g_i (note that this immediately creates a cycle). This edge is called a *group indirect reads_before* edge.

3. For each object x, G embodies a total order over all groups that write x, i.e., for each pair of groups g_i and g_j that write x, there is either a path in G from g_i to g_j or from g_j to g_i. This total order is called a *group write order* for x, and is denoted $\Rightarrow_x^g$.

4. For each object x and groups g_i, g_j, g_k such that g_j *reads_x_from* g_i and $g_i \Rightarrow_x^g g_k$, G contains a path from g_j to g_k. This path is called a *group reads_before* path.

Reads_from edges (henceforth, we will refer to group edges simply as reads_from, indirect reads_before, etc., unless an ambiguity may arise) capture the conflicts arising from reads_from relations between different groups; while indirect reads_before edges capture conflicts resulting from a group reading an object x from another group g, where the value read is not the final value of x with respect to g with order $<_g$. However, these edges are not enough to capture all group conflicts, and hence to establish the one-copy serializability of all transaction executions. Therefore $1{-}SGG[L]$ must also have enough edges to embody, for each object x, a total order on all groups that write x. Furthermore, $1{-}SGG[L]$ must contain all reads_before paths resulting from the write orders chosen.

Note that if each group contains one transaction only, then $1{-}SGG[L]$ degenerates to the logical serialization graph defined in [Bernstein87]. In [Bernstein87] it is proved that the existence of an acyclic logical serialization graph is a necessary and sufficient condition to ensure one-copy serializability. With our model, the existence of an acyclic $1{-}SGG[L]$ does not guarantee one-copy serializability. Each group is a set of transactions that may not be serializable. However, we can show that:

Theorem 1 Let L be a log over a set of transactions T. L is one-copy serializable if and only if T can be partitioned into groups such that:

1. Each group g is one-copy serializable, with order $<_g$.

2. There is a corresponding acyclic $1{-}SGG[L]$.

Proof: The proof is fairly detailed and can be found in [El Abbadi87].

Theorem 2 A group concurrency control protocol ensures one-copy serializability.

Proof: The proof can be found in [El Abbadi87].

5. Analysis of Protocols

We will show that several known concurrency control protocols [Gifford79, Eager83, Skeen84, El Abbadi86, Herlihy87] are instances of our group paradigm. In this section we show that they all partition transactions into groups, where each group uses a local policy to ensure the one-copy serializability of all transactions in that group. Furthermore, each protocol uses a global policy that ensures a group order on all groups, and hence, from Theorem 2 ensures one-copy serializability.

Little work has been done comparing the level of data availability attained by concurrency control protocols or the costs they incur in a database that may suffer from site and communication failures. Coan et al. [Coan86] analyze the availability level attained by different protocols assuming (1) that each object is fully replicated, that is, each object has a copy on every site, and (2) that partitioning failures are limited to the single case where the network is divided into two partitions only. Using these strong assumptions, a maximum availability level is defined, and it is shown that most of the protocols discussed attain this maximum level of availability. In this section we compare concurrency control protocols in the way they achieve correctness, and we make no assumptions about object replication or about how failures may partition the database.

We start our analysis with Gifford's quorum protocol [Gifford79], where all transactions are members of one group only. This is followed by the missing write protocol [Eager83], where transactions use two different local policies, and the class conflict protocols [Skeen84], which only handles the case where the network partitions (and no recovery takes place). We then show that both the accessibility threshold protocol [El Abbadi86] and Herlihy's quorum consensus protocol [Herlihy87] are instances of the group paradigm.

5.1. Gifford's Quorum Protocol

The *quorum protocol* [Gifford79] presented by Gifford, maintains correctness by requiring all logically conflicting operations to physically conflict. This protocol represents a degenerate case of the paradigm, where all transactions are members of the same group, and use the same local policy. This policy associates with each object x two integers: the *read quorum*, $q_r[x]$, and the *write quorum*, $q_w[x]$. The read and write quorums determine the number of copies accessed or written by a logical read or write operation. Associated with each copy is a *version number*, which is initialized to 0, and incremented by write operations. Furthermore, all transactions use a conflict preserving concurrency control protocol. The local policy requires each set of size $q_w[x]$ to have at least one copy in common with any set of size $q_r[x]$, or $q_w[x]$. (Thomas [Thomas79] describes a special case of this protocol, where read and write operations are required to access and write a majority of copies.)

Since all transactions use the same local policy, transactions form one group only, and the conditions on the global policy are trivially satisfied. It is also straightforward to prove that the local policy ensures that the group is one-copy serializable. Informally, the restrictions on quorums ensure that any two logically conflicting operations must also physically conflict. Since all transactions use a conflict preserving concurrency control protocol, one can show that the group is one-copy serializable. A more detailed proof is given in [Bernstein83].

5.2. The Missing Write Protocol

In this section, we present a simplified version of the *missing write protocol* [Eager83] presented by Eager and Sevcik. This protocol is a special instance of the group paradigm where transactions use two kinds of local policies: the *normal mode policy*, and the *failure mode policy*. The normal mode policy executes transactions using a read-one, write-all approach. The failure mode policy executes transactions using read and write quorums satisfying the Gifford quorum conditions. Both policies require transactions to use a conflict preserving concurrency control protocol. A *normal* (*failure*) *group* is defined as all transactions using the normal (failure) mode policy. Hence, as we proved in the previous section, each group is one-copy serializable.

The global policy imposes a group order $<$ on these two groups such that the normal group precedes the failure group. The global policy is enforced as follows. A transaction t_i starts using the normal mode policy, and if it cannot executed a physical write operation on a copy of object x (due to site or partitioning failures), it switches to the failure mode policy. Assume that t_i was writing object x. The protocol requires that t_i leave a *missing write token* for x at all copies that it accesses or writes. When a normal mode transaction encounters any missing write token, it also switches to failure mode and leaves missing write tokens for all copies that it is aware of at all copies it accesses or writes.

Now we show that the global policy ensures conditions 2.a, 2.b, and 2.c. We only have to consider the cases when a transaction in the failure group reads from a transaction in normal group (later we will briefly discuss how recovery from failures takes place).

(a) Any transaction in the failure group g_f either reads the value written by another transaction in the same group, or in the normal group g_n. But, by definition, the normal group precedes the failure group. Hence, if g_f *reads_x_from* g_n, then $g_n < g_f$.

(b) If g_f *reads_x_from* g_n, then by definition, there is no third group that is ordered between the normal and the failure groups.

(c) Finally, let g_f *reads_x_from* g_n. Any failure group transaction t_f leaves missing write tokens at a read quorum of x. Since any normal group transaction writes all copies of x, it must encounter at least one such token, and hence, will switch to the failure mode policy. Therefore, once t_f is executed, no other normal group transactions are executed. Furthermore, t_f reads the value with the highest version number, and hence, it reads the final value of x with respect to the local order $<_{g_n}$ on all transactions in the normal group.

Once sites and communication links recover, a special *copier* transaction is executed at all sites that performs the following: (1) For each copy x_s that missed some write operations, the copier transaction accesses a failure mode read quorum of copies and writes x_s with the value of the copy with the highest version number. (2) After x_s is updated, the missing write tokens of x_s are deleted from all sites. (In [Eager83], some of the updates can be executed during the execution of the user transactions to increase efficiency and to avoid unnecessary execution of transactions using the failure mode policy.) Subsequent to failures and recovery, the copier transactions "logically" reinitialize the database to ensure that all transactions use the normal mode policy, and hence, form a new normal group. Therefore, groups alternate in pairs between normal and failure.

5.3. Class Conflict Protocol

In the *class conflict protocol* proposed by Skeen and Wright [Skeen84], transactions are *a priori* divided into *classes* [Bernstein80], which may be well defined transaction types or may be syntactically defined by the objects a transaction may read or write. It is assumed that failures can be detected correctly whenever they occur, and when a partitioning failure occurs, each partition must decide which classes of transactions it will allow to execute.

All transactions executing in a partition form a group, with all transactions executing in the initial configuration forming a group g_0. Transactions executing in a partition use any correct concurrency control protocol, and a read-one write-all (copies in partition) protocol. Hence, each group is one-copy serializable.

The global policy of this protocol uses *class conflict graphs* to determine which transactions to allow to execute in each partition. This graph corresponds to a $1\text{–}SGG[L]$, where nodes are classes of transactions, and edges correspond to potential logical conflicts between classes. Unlike the $1\text{–}SGG[L]$, which is used for proof purposes only, in the class conflict protocol each partition constructs a class conflict graph containing all classes that it may execute as well as classes that may be executed in other partitions.

When a partitioning failure occurs, each partition is assigned a set of classes that could be executed in this partition. Based on this assignment, each partition makes a worst case assumption concerning which classes may be executed in other partitions and builds a class conflict graph. Each partition analyzes its class conflict graph, and checks whether the graph contains any cycles that span more than one partition (called *multi-partition cycles*). If any such cycles exist, the partition must delete enough classes until the graph contains no multi-partition cycles.

As presented, this protocol does not discuss what actions should be taken after recovery from partitioning. Hence, the only case handled is when the system is initially connected and then failures occur that lead to partitioning. The class conflict graph ensures a total order on all classes executing in different partitions, hence, a total order $<$ on all groups can be defined to be consistent with that order, with group g_0 preceding all other groups. We now show that the global policy satisfies conditions 2.a, 2.b, and 2.c of the group paradigm:

(a) Transactions in g_0 only read values written in g_0. Any transaction t_j in group g_j reading object x, either reads the value written by another transaction in the same group g_j or in g_0, which precedes g_j. Hence, if g_j *reads_x_from* g_i, it must be the case that i=0, and therefore $g_i < g_j$.

(b) Let g_j *reads_x_from* g_0. Assume that g_k writes x, where $g_0 < g_k < g_j$. Since $g_k < g_j$, then the class conflict graph contains a path from g_k to g_j. Furthermore, since g_k writes x, and g_j *reads_x_from* g_0, then any class conflict graph must contain an edge from g_j to g_k (this edge is the result of a potential conflict between a transaction in g_j that reads object x, which may be written by a transaction executed in g_k). Hence, the class conflict graph contains a cycle, contradicting the protocol, which ensures no multi-partition cycles. Therefore, if g_j *reads_x_from* g_0, then there can be no g_k that writes x, where $g_0 < g_k < g_i$.

(c) Let g_j *reads_x_from* g_0. Every write operation in g_0 writes all copies of x; hence, if g_j reads x, it must read the final value of x in g_0 with respect to $<_{g_0}$.

5.4. The Accessibility Threshold Protocol

In the Accessibility Protocol presented in [El Abbadi86], instead of the static quorums used in Gifford's protocol, the system designer is given the flexibility of changing the read and write quorums as appropriate during system execution. A site s has associated with it a special set of sites called its *view*, which contains all sites s assumes it can communicate with. Each view has a *view_id*, and two sites are said to have the *same* view if their views have the same view_id. View-ids form a total order. Each object x is assigned *read* and *write accessibility thresholds* $A_r[x]$ and $A_w[x]$. An object is *read (write) accessible* in a view v if $A_r[x]$ ($A_w[x]$) copies of x reside on sites in v. In a view v, each accessible object x is assigned *read* and *write* quorums $q_r[x,v]$ and $q_w[x,v]$. For a transaction to execute, it picks a view v, uses the quorum assignments of v for the translation of its logical operations. Version numbers are used in the standard way [Gifford79].

All transactions executing in the same view form a group. The local policies ensure the one-copy serializability of each group. In each view, v, the *read* and *write* quorums assigned to an object x, $q_r[x,v]$ and $q_w[x,v]$ respectively, must satisfy Gifford's conditions on quorums: for an object x, a set of size $q_w[x,v]$ intersects with any set of size $q_w[x,v]$ or $q_r[x,v]$. A transaction executing in view v executes a read of x by accessing $q_r[x,v]$, and executes a write of x by writing $q_w[x,v]$. Furthermore, a transaction executing in view v is restricted to access or write copies that reside on sites with view v. These conditions, in

addition to a conflict preserving concurrency control protocol ensure that each group is one-copy serializable.

The unique view_ids associated with views form a total order, thus defining a total order $<$ on all groups. Hence, each group is uniquely identified by a view and a view-id. The global policy uses the following two mechanisms.

1. **The Accessibility Threshold Restrictions.** Each object x is assigned *read* and *write accessibility thresholds* $A_r[x]$ and $A_w[x]$. The sum of the read and write accessibility thresholds for each object x, i.e., $A_r[x]+A_w[x]$, must exceed the number of copies of x. Furthermore, write quorums of an object x must always be greater than the write threshold for object x, i.e., $q_w[x,v] > A_w[x]$ for all views v. An object is *read (write) accessible* in a view v if $A_r[x]$ ($A_w[x]$) copies of x reside on sites in v. A transaction executing in view v may only read (write) an object if it is read (write) accessible in v.

2. **The Update Transaction.** A site s changes its view to v (with view-id v_id) by executing an *update transaction.* For all read accessible objects x in v with copies residing on s, the update transaction accesses $A_r[x]$ copies of x, reads the value with the highest version number, and updates the local copy x_s with that value. An access operation is rejected by any site with a view-id greater than v_id. If all access operations are successful, view v is *installed* at s. When a site is accessed by an update transaction installing v, it *inherits* v, by trying to change its view to v by executing an update transaction. If it fails it tries to installs a view with a higher view_id.

We now show that the global policy satisfies conditions 2.a, 2.b and 2.c of the group paradigm. For the purposes of the proof, an update transaction t, executed during the installation of view v_j, is a member of group g_j, the set of all user transactions executing in v_j. Note that user transactions are restricted to read copies residing on sites in the same view, hence, update transactions are the only transactions allowed to access copies written by transactions in a different group. Therefore, we only have to show that update transactions satisfy the conditions 2.a, 2.b and 2.c.

(a) An update transaction t_j installing a view v_j with view-id v_j_id (and hence a member of some group g_j) is restricted to read copies residing on sites whose views have view-ids less than or equal to v_j_id, thus that if t_j *reads_x_from* t_i where $t_i \in g_i$ and $t_j \in g_j$, then $g_i < g_j$.

(b) Consider an update transaction t_j installing a view v_j with view-id v_j_id. Update transaction t_j must access $A_r[x]$ copies of an object x to update some copy of read accessible object x. The restrictions on write accessibility thresholds and write quorums ($q_w[x,v] > A_w[x]$), as well as on read and write accessibility thresholds ($A_r[x]+A_w[x] > n[x]$) ensure that update transaction t_j (in group g_j) accesses at least one copy in any set of size $q_w[x]$. But every site s accessed by t_j tries to inherit view v_j, and if the installation process fails s installs a view with a higher view-id. Hence, once t_j has accessed $A_r[x]$ copies of x, every set of copies of x of size $q_w[x]$ has at least one copy residing on a site whose view has view-id v_j_id or greater. Since every write operation executing in a view v writes $q_w[x]$ copies that reside on sites with view v, once t_j is executed, no write operation can be executed on object x in a view with view-id less than v_j_id. Transaction t_j reads the copy with the highest version number of $A_r[x]$ copies and each write x writes at least $A_w[x]$, hence, the value read by t_j is the most up-to-date value, i.e., if g_j reads x, which was written in g_i, then (1) there is no g_k that writes x such that $g_i < g_k < g_j$, and (2) g_j reads the final value of x in g_i with respect to local order $<_{g_j}$, thus satisfying conditions 2.b and 2.c of the global policy.

5.5. Herlihy's Quorum Consensus Protocol

Herlihy [Herlihy87] presents a generalization of Gifford's quorum protocol for multi-version databases, where a transaction chooses a natural number, called its *level*, to execute in. Each object x has a *quorum assignment table*, which assigns for each level l, a read quorum $q_r[x,l]$ and a write quorum $q_w[x,l]$. Initially, each level has an original quorum assignment binding with an initial timestamp. Subsequently, the quorum assignment bindings may be changed, in which case the timestamp associated with the level is updated to reflect the new binding. Each copy stores a sequence of timestamped versions, one for each level containing the most recently written version, if any. Each transaction is assigned a unique timestamp that determines its serialization order. A transaction t executing in level l writes an object x by writing the l^{th} version of $q_w[x,l]$ copies of x. Each version written is assigned t's timestamp. A transaction t

executing in level l reads an object x by accessing $q_r[x,l]$ copies of x. For each accessed copy t accesses the version associated with the highest level less than or equal to l, and reads the copy with the highest timestamp.

This protocol fits neatly into our paradigm. All transactions executing at the same level, and that use the same quorum assignment binding form a group. All groups use a local policy that synchronizes transactions by a mechanism capable of assigning logical timestamps to transactions so that their timestamp ordering reflects their serialization order [Reed83]. Hence, each group is one-copy serializable.

A group is uniquely identified by the pair $<l,ts>$, where l is the level transactions execute in, and ts is the timestamp of the quorum binding. By definition, both levels and timestamps are totally ordered, hence, there is a group order $<$ on the set of all groups as follows. For $g_i=<l_i,ts_i>$ and $g_j=<l_j,ts_j>$, $g_i<g_j$ if $l_i<l_j$ or if $l_i=l_j$ and $ts_i<ts_j$. The global policy uses the following mechanisms.

1. **The Ratchet Lock and Quorum Assignment Bindings.** Each copy has a *ratchet lock*, which is a counter that records the highest level of a transaction that has read that copy. A copy rejects write requests from any transaction whose level is less than the copy's ratchet lock. Writes are said to be *enabled* for an object at level n if none of its ratchet locks exceed n. Furthermore, a copy with quorum assignment binding ts for level n rejects all operations executed at level n but with quorum assignment bindings $ts'<ts$.

2. **The Quorum Intersection Invariant:** Quorum assignments are required to satisfy the *quorum intersection invariant.* If writes are enabled to object x at level l, then any set of size $q_w[x,l]$ must have at least one copy in common with any set of size $q_r[x,m]$, for all levels $m \geq l$.

3. **The Deflation Transaction.** A copy changes the quorum assignment binding of object x for level m from $q_r[x,m]$ and $q_w[x,m]$ (called *old read* and *write quorums*) to $q_r[x,n]$ and $q_w[x,n]$ (called *new read* and *write quorums*), where n is the same level as m but with the new quorum assignment binding, by atomically executing a *deflation transaction.* The deflation transaction:

   (1) Accesses $q_r[x,m]$ copies of x, and reads the closest preceding version for level m with the highest timestamp.

   (2) Disables write operations at all levels $l<m$ with write quorums $q_w[x,l]$, where a set of size $q_w[x,l]$ does not intersect with a set of size $q_r[x,n]$. This is done by advancing the copies' ratchet locks beyond m.

   (3) Writes the read version and its timestamp to $q_w[x,n]$ copies of x.

   (4) Writes the highest ratchet lock to $q_r[x,n]$ copies of x.

   (5) Updates the quorum assignment tables and their associated timestamp bindings, at enough copies (called *read* and *write coquorum*), so that these updated copies have a nonempty intersection with every set of size $q_r[x,m]$ and $q_w[x,m]$, the old quorums.

For the purposes of the proof, if deflation transaction t_d *reads_x_from* t, where t is a member of g, then t_d is considered a member of g. We now show how this global policy satisfies the conditions of the group paradigm:

(a) By definition, a read operation executed in group g_j (where $g_j=<l_j,ts_j>$ always reads the value of an object x written in g_j or in a group g_i that precedes g_j in group order $<$, thus satisfying condition 2.a of the global policy.

(b) The ratchet locks and timestamp binding, in addition to the quorum intersection invariant, ensure that once g_j reads x using an original quorum assignment, no write operation is allowed to execute in any group g_k that precedes g_j in $<$. If g_j does not have an original quorum assignment, then a sequence of deflation transactions must have been executed, and a simple inductive argument also shows that once g_j reads x, no write operation is allowed to execute in any group g_k that precedes g_j in $<$. Since g_j reads the the closest preceding version for level l_j, there is no g_k that writes x such that $g_i<g_k<g_j$, and the value read by g_j is the final value of x in g_i, thus satisfying conditions 2.b and 2.c of the global policy.

We note the similarities between the methods used by the global and local policies. In the local policy, which is essentially a timestamp-based protocol [Reed83], the timestamp order imposes a total order on all transactions in a group; in the global policy the pair <level, timestamp> imposes a total order all groups. In the local policy, a read of x reads the version written by the closest preceding transaction in timestamp order; in the global policy, a read of x reads the version written by the closest preceding group in the group order. Finally in the local policy, once a read operation is executed, a read-timestamp disallows write operations with lower timestamps; in the global policy, once a read operation is executed, a ratchet lock disallows write operations from executing in

preceding groups in the group order.

6. Conclusion

We presented a paradigm for describing and proving the correctness of concurrency control protocols for distributed replicated databases. This paradigm presents a framework for both developing and analyzing concurrency control protocols, especially those that are designed to handle partitioning failures. In contrast to previous models for concurrency control [Bernstein87, Herlihy87], which concentrated exclusively on the issue of correctness, our paradigm provides a modular approach for analyzing protocols as well as proving them correct. Instead of serializing all transactions using one "policy", our paradigm divides the problem into two tasks; first serializing transactions in each group using a local policy, and then serializing all groups using a global policy. Our approach provides a unifying paradigm for understanding several protocols that have appeared in the literature, thus showing how similar they are. The correctness of these protocols is a simple consequence of the fact that they are all different instances of our group paradigm. In fact, they all satisfy the local and global serialization requirements. They differ in the implementation and the cost associated with the execution of operations.

7. References

[Bernstein80] Bernstein, P., Shipman, D., and Rothnie, Jr., J., "Concurrency Control in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems 5*, 1 (March 1980), 18-51.

[Bernstein83] Bernstein, P., and Goodman, N., "The Failure and Recovery Problem for Replicated Databases," *Proc. 2nd ACM Symp. on Princ. of Distributed Computing*, Montreal, Quebec (August 1983), 114-122.

[Bernstein87] Bernstein, P., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, Reading, Massachusetts 1987.

[Coan86] Coan, B., Oki, B. and Kolodner, E., "Limitations on Database Availability when Networks Partitions," *Proc. 5th ACM Symp. on Princ. of Distributed Computing*, Calgary, Alberta, Canada (August 1986) 63-72.

[Davidson85] Davidson, S., Garcia-Molina, H., and Skeen, D., "Consistency in Partitioned Networks," *Computing Surveys 17*, 3 (September 1985) 341-370.

[Eager83] Eager, D., and Sevcik, K., "Achieving Robustness in Distributed Database Systems," *Transactions on Database Systems 8*, 3 (September 83), 354-381.

[El Abbadi86] El Abbadi, A. and Toueg, S., "Availability in Partitioned Replicated Databases," *Proc. 5th ACM Symp. on Princ. of Database Systems*, Cambridge, Massachusetts (March 1986), 240-251.

[El Abbadi87] El Abbadi, A. "A Paradigm for Concurrency Control Protocols for Distributed Databases," TR 87-853 (Ph.D. Thesis), Department of Computer Science, Cornell University (August 1987).

[Eswaran76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I., "The Notions of Consistency and Predicate Locks in a Database System," *Comm. of the ACM 19*, 11 (November 1976), 624-633.

[Gifford79] Gifford, D., "Weighted Voting for Replicated Data," *Proc. of the 7th Symposium on Operating Systems Principles*, (December 1979).

[Herlihy87] Herlihy, M., "Dynamic Quorum Adjustments for Partitioned Data," *Transactions on Database Systems 12*, 2 (June 87), 170-194.

[Moss82] Moss, E., "Nested Transactions and Reliable Distributed Computing," *2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*, (1982).

[Reed83] Reed, D.P., "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems 1*, 1 (February 1983) 3-23.

[Skeen84] Skeen, D., and Wright, D. "Increasing Availability in Partitioned Networks," *Proceedings of the 3rd ACM-SIGMOD Symposium on Principles of Database Systems*, New York (April 84) 290-299.

[Thomas79] Thomas, R., "A Majority Consensus Approach to Concurrency Control," *ACM Transactions on Database Systems 4*, 2 (June 1979), 180-209.