VLDB Journal, 2, 215–241 (1993), Andreas Reuter, Editor

Supporting Consistent Updates in Replicated Multidatabase Systems

Weimin Du, Ahmed K. Elmagarmid, Won Kim, and Omran Bukhres

Received November 7, 1990; revised version received January 26, 1992; accepted December 16, 1992.

Abstract. Replication is useful in multidatabase systems (MDBSs) because, as in traditional distributed database systems, it increases data availability in the presence of failures and decreases data retrieval costs by reading local or close copies of data. Concurrency control, however, is more difficult in replicated MDBSs than in ordinary distributed database systems. This is the case not only because local concurrency controllers may schedule global transactions inconsistently, but also because local transactions (at different sites) may access the same replicated data. In this article, we propose a decentralized concurrency control protocol for a replicated MDBS. The proposed strategy supports prompt and consistent updates of replicated data by both local and global applications without a central coordinator.

Key Words. Multidatabases, replicated data management, concurrency control, replica control, serializability, resolvable conflicts.

1. Introduction

A multidatabase system (MDBS) is a federation of pre-existing database systems (called local database systems, or LDBSs). An MDBS is the natural result of the shifting priorities and needs of an organization as it acquires new database systems that have been designed independently. For many applications, an MDBS is an attractive alternative to a single distributed database system in that it allows existing software developed for each LDBS to continue to be executable without modification. The most important feature of an MDBS is the autonomy of its LDBSs. Local autonomy is both desirable and necessary; it facilitates flexible interconnection of various kinds of LDBSs, ensures consistency and security of LDBSs, and guarantees that old applications are executable after interconnection.

Weimin Du, Ph.D., is Technical Staffmember, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304; Ahmed K. Elmagarmid, Ph.D., is Associate Professor and Omran Bukhres, Ph.D., is Visiting Associate Professor, Computer Sciences Department, Purdue University, W. Lafayette, IN 47907; Won Kim, Ph.D., is President and CEO, UniSQL, Inc., 9390 Research Blvd., Austin, TX 78759.

The main objective of interconnecting otherwise isolated LDBSs is the sharing of data. An effective approach to data sharing among LDBSs is the replication of local data at remote sites. Replication is desirable in MDBSs because it allows a local user to read otherwise "remote" data easily, thereby reducing retrieval costs. Replication also increases data availability in the presence of failures. On the other hand, replication adds complexity to transaction processing. For example, updates to replicated data at one site must be propagated to other sites so that mutual consistency is maintained.

In traditional replicated database systems, mutual consistency is guaranteed by executing transactions in a one-copy serializable manner (Bernstein et al., 1987). In one-copy serializable executions, transactions see the same database states as in a serializable execution of the same set of transactions on a single copy (i.e., non-replicated) database. One-copy serializability is ensured by a concurrency control protocol which guarantees that physical data (regardless of replication) are accessed consistently (e.g., in a serializable fashion), and by a replica control protocol that ensures that physical copies of all replicated data are accessed consistently. For example, a transaction may read any copy of a replicated data item. To update replicated data, however, it must write all copies.

Although the basic paradigm remains largely unchanged, concurrency control in replicated MDBSs presents significantly increased challenges. Ensuring such control is more difficult than in traditional databases because the component LDBSs schedule local executions independently and possibly inconsistently. It is also more difficult than in non-replicated MDBSs because both global and local applications can update replicated data. Furthermore, due to the constraints of autonomy and availability, there can be no central agent coordinating such concurrent accessing.

Quasi-serializability has been proposed as an alternative to the traditional serializability approach (Du and Elmagarmid, 1989). Quasi-serializability reflects the fact that transactions are scheduled by LDBSs independently. Instead of trying to ensure consistent serialization order at different sites, it requires only consistent execution order for global transactions. It has been shown that quasi-serializability is easier for the global transaction manager (GTM) to ensure, and possible inconsistencies can be resolved by controlling information flow between global subtransactions.

In this article, we first extend the basic quasi-serializability theory to handle replicated data, resulting in one-copy quasi-serializability (1QSR). The main result is a concurrency control protocol for a replicated MDBS that supports prompt and consistent (i.e., one-copy quasi-serializable) updating of replicated data in a decentralized and autonomous fashion. The protocol is decentralized; there is no central agent imposed to coordinate access to replicated data. Instead, a set of local servers is used to coordinate the execution of transactions that access replicated data. Each server manages submissions of global and local transactions at each site independently, without consulting the global transaction manager and other servers. The proposed strategy maintains the global consistency (i.e., 1QSR) of replicated multidatabases in the presence of the concurrent execution of both local and global

transactions. Another advantage of the protocol is that it does not violate local autonomy; the LDBSs have full control over their data regardless of replication. A local transaction independently updates replicated data that logically belong to its host LDBS. It can also read other replicated data that have copies at its site. In addition, no modification to the LDBSs is required, and local transactions that do not update replicated data are processed in the same way as if the LDBSs were not interconnected.

The rest of this article is organized as follows. In Section 2 some basic concepts and terminology are introduced. An overview of replicated data management in multidatabase systems is then given in Section 3, showing that the crux of the problem is to devise a decentralized concurrency control protocol. Basic quasi-serializability is extended in Section 4 to handle replicated data. In Section 5 we present a concurrency control protocol that supports prompt and consistent access to replicated data by both local and global applications. In Section 6 several related issues are discussed. Section 7 contains concluding remarks.

2. System Model

A replicated MDBS consists of a set of LDBSs, $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ and a set of transactions, $\mathcal{T}$.[1] Each local database system is defined by a pair of data sets, $S_i = \langle \mathcal{D}_i, \mathcal{P}_i \rangle$, where $\mathcal{D}_i$ is a set of *logic data items* that belong to S_i, while $\mathcal{P}_i$ is a set of physical data copies that reside at S_i. We also use $\mathcal{D} = \cup_{i=1}^{n} \mathcal{D}_i$ and $\mathcal{P} = \cup_{i=1}^{n} \mathcal{P}_i$ to denote respectively the set of logical data items that belong to the set of physical copies that reside at the MDBS. The function $f : \mathcal{D} \rightarrow 2^{\mathcal{P}}$ (where $2^{\mathcal{P}}$ denotes the powerset of $\mathcal{P}$) gives for each data item, d, the set of data copies which implement d. A data item, d, is said to be replicated if the cardinality of $f(d)$ is greater than one. Otherwise, it is said to be non-replicated. For a replicated data item, $d_i \in \mathcal{D}_i$, there always exists a data copy, $d_i^i \in \mathcal{P}_i \cap f(d_i)$ (i.e., the copy that resides at its host site), which is called the *primary copy*.[2]

There are two kinds of transactions in a replicated MDBS: *global* and *local* transactions. Local transactions represent those applications developed on each LDBS before the construction of the MDBS and which therefore access $\mathcal{D}_i$ only. Global transactions represent applications developed afterwards which usually access multiple $\mathcal{D}_i$'s. Local transactions can be further differentiated between basic local transactions which update non-replicated data only, and *replica-update* local transactions which update both replicated and non-replicated data. From a user's point

1. Notation: Calligraphic letters denote sets, Roman letters denote acronyms, and italic letters denote instances; lower case letters are used for data items and capitals for transactions and local database systems.

2. Notation: We use d_i^j to denote the physical copy of data item d_i at site S_j; the subscript specifies the data item and the superscript specifies the site.

of view, replica-update and basic local transactions present the same appearance, because they access only data items that logically belong to a single LDBS. From a transaction management point of view, however, replica-update transactions appear to be global, because they access data copies that physically reside at several sites. In the rest of this article, those two situations will be simply referenced as local and replica-update transactions.

Formally, transaction set $\mathcal{T}$ consists of $2n+1$ subsets, $\mathcal{GT}, \mathcal{LT}_1, \mathcal{RT}_1, \mathcal{LT}_2, \mathcal{RT}_2, ..., \mathcal{LT}_n, \mathcal{RT}_n$, where $\mathcal{GT}$ is the set of global transactions, $\mathcal{LT}_i$ is the set of local transactions belonging to S_i, and $\mathcal{RT}_i$ is the set of replica-update transactions belonging to S_i. A global transaction $\mathcal{G}_i$ consists of a set of subtransactions (called *global subtransactions*) $G_i^1, G_i^2, ..., G_i^n$, where $\mathcal{G}_i^j$ accesses $\mathcal{P}_j$ only.[3] Similarly, a replica-update transaction, R_i, consists of a *replica-update subtransaction*, R_i^i (that accesses $\mathcal{P}_i$ only), and a set of *replica-propagation subtransactions*, $R_i^1, R_i^2, ..., R_i^n$, where R_i^j accesses only $f(D_i) \cap \mathcal{P}_j (i \neq j)$. We use $\mathcal{RT} = \cup_{i=1}^n \mathcal{RT}_i$ and $\mathcal{LT} = \cup_{i=1}^n \mathcal{LT}_i$ to denote, respectively, the set of all replica-update transactions and the set of all local transaction in the MDBS. We also use $\mathcal{T}_i = \mathcal{LT}_i \cup \{T_0^i | T_0 \in \mathcal{GT}\} \cup (\cup_{j=1}^n \mathcal{RT}_j)\}$ to denote the set of transactions that are executed at site S_i.

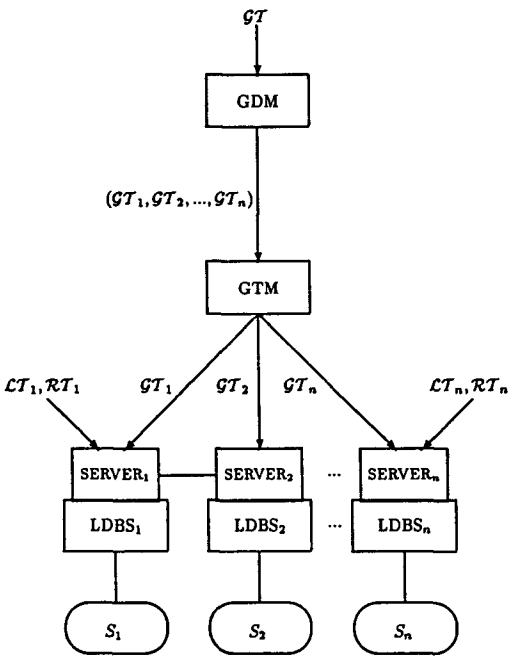
Generally, a transaction is a finite set of partially ordered operations. Each is either a read or a write operation. In this paper, we use $r_t(d_0^s)$(or $w_t(d_0^s)$) to denote the read (or write) operation of transaction T reading (or writing) a copy of data item d_0 at site S_s. The set of all data copies read (or written) by T is denoted as $\mathcal{R}(T)$ (or $\mathcal{W}(T)$). $\mathcal{O}(T) = \mathcal{R}(T) \cup \mathcal{W}(T)$ is the set of all data copies accessed by T.

For $G \in \mathcal{GT}$, $R_i \in \mathcal{RT}_i$ and $L_i \in \mathcal{LT}_i$, accessibility to data copies is defined as follows:

1. $\mathcal{R}(G) \subseteq \mathcal{P}$
2. $\mathcal{W}(G) \subseteq \mathcal{P}$
3. $\mathcal{R}(R_i^i) \subseteq \mathcal{P}_i$
4. $\mathcal{W}(R_i^i) \subseteq \mathcal{P}_i \cap f(\mathcal{D}_i)$
5. $\mathcal{R}(R_i^j) = \emptyset$
6. $\mathcal{W}(R_i^j) = \mathcal{P}_j \cap f(\mathcal{D}_i)$
7. $\mathcal{R}(L_i) \subseteq \mathcal{P}_i$
8. $\mathcal{W}(L_i) \subseteq \{p \in \mathcal{P}_i | \exists d \in \mathcal{D}_i$ such that $f(d) = \{p\}\}$

3. Notation: We use T_i^j to denote the subtransaction of T_i at site S_j.

Figure 1. Transaction Processing in Replicated MDBSs



The transaction processing model used in this article is presented in Figure 1. There are three major components: a global data manager (GDM), a global transaction manager (GTM), and a set of servers residing at each local site.

Both the GDM and the GTM are centrally located, while a server is superimposed on the LDBS at each site. The interrelationship of these components is as follows. A global transaction (in logical operations) is first submitted to the GDM, which decomposes it into a set of subtransactions (in physical operations) and passes them to the GTM. The GTM then sends the subtransactions to servers at local sites and coordinates their execution. Each server in turn sends each operation of the subtransaction (including commit/abort) to the underlying database as instructed by the GTM.

In this model, local and replica-update transactions first must be submitted to the server, rather than directly to the underlying database. The server translates each logical operation on a data item into the corresponding physical operation on the local copies of the data and forwards it to the underlying database. The server also propagates each replica-update operation to servers at remote sites so that mutual consistency is maintained. Clearly, local and replica-update transactions must be submitted to the server to maintain mutual consistency, although it may imply modifications to pre-existing local applications.

3. Replicated Data Management for an MDBS

The ultimate goal of mutual consistency carries with it two implications. First, as in non-replicated databases, data items (regardless of replication) must be accessed in a consistent order (*data consistency*). For example, two transactions may not update two data items in different orders. Second, physical copies of the same replicated data must be accessed by different transactions in a consistent order (*replica consistency*). Therefore, two transactions may not update the same replicated data in different orders at two sites.

These two restrictions are accommodated by a pair of separate protocols. A *replica control protocol* ensures replica consistency by translating each logical operation into one or more physical operations, while a *concurrency control* protocol ensures data consistency by coordinating the execution of physical operations. Together they maintain mutual consistency of a replicated DBMS. In this section, we review the traditional strategies for both protocols to identify key issues in extending the strategies for a replicated MDBS. First, let us summarize the basic requirements and assumptions of replicated data management in an MDBS.

3.1 Assumptions and Requirements

The following requirements are desirable for a replicated data management protocol in a replicated MDBS: preserving *local autonomy*; supporting *consistent access* to replicated data; enhancing *system reliability* and *data availability*; and providing *transparent access* to replicated data. These requirements and their implications will now be discussed in further detail.

Local Autonomy. We assume that local databases are autonomous. Local autonomy reflects the fact that local databases were independently designed and administrated. Data were replicated to facilitate sharing. We make no assumption regarding the accessibility of local data. In other words, local data are fully controlled by their host LDBSs. These conditions generate two implications. First, no modification to the local DBMSs (including local transaction managers) is allowed. Second, it is impossible, or at least undesirable, to impose a central agent that regulates access to all replicated data. The LDBSs should be able to independently update each data item (regardless of replication), and each should have full control over updates to its data by remote transactions.

Consistency. The conventional correctness criterion for data management is serializability (Bernstein et al., 1987). Ideally, we would like to devise a protocol that ensures the serializability of global executions as long as local DBMSs ensure local serializability. However, this seems to be difficult to achieve given the autonomy requirement (Du et al., 1989). The protocol we propose ensures serializability only if all LDBSs ensure strict serializability (Papadimitriou, 1986). In general, however, the protocol ensures only a weaker notion of consistency: quasi-serializability (Section 4).

Reliability and Availability. To facilitate effective data sharing, local users should be able to read replicated data both independently (without consulting other sites) and efficiently (reading the nearest copy only). In addition, they should always be able to read local data (regardless of replication), even when the site is isolated. Similarly, local users should be able to update local data independently. Although updates must be propagated to remote sites, local updates should be committed as soon as possible, without waiting for the completion of remote ones. This implies that any protocol (e.g., two-phase commit) that aborts replica-update transactions due to inconsistent update propagations violates the availability requirement. Two-phase commitment is neither desirable nor necessary for replica-update transactions (Section 6).

Replication Transparency. We assume that, in replicated MDBSs, local users are not aware of the replication of local data. As a matter of fact, for those local applications developed before the databases were integrated, all data are local and non-replicated. It is therefore necessary to provide transparent access to replicated data. Local applications, regardless of the data they access, should be treated uniformly. This has led to the distinction between global and replica-update transactions. In our transaction processing model (Figure 1), replica-update transactions are submitted and executed in the same way as other local transactions and are not controlled by the GTM. The server, however, must distinguish between replica-update and local transactions and propagate any updates made by replica-update transactions. This requires that each server keep a replicated data dictionary for data at the site. In addition, a global dictionary is maintained and used by the GDM for replicated data at all sites.

In the next two subsections, we shall discuss the implications of the above requirements on replica and concurrency control for a replicated MDBS.

3.2 Replica Control

In the absence of failures, replication control simply involves the translation of each pair of logically conflicting operations into physical operations conflicting on at least one physical copy. As a special case, the read-one-write-all protocol translates each logical read operation into one physical read operation and each logical write operation into a set of all physical write operations. By reading only the nearest copy, the algorithm implements the efficient reading of a data item, an attractive feature when reads outnumber writes, as in most real-world applications (Stonebraker, 1988).

In the presence of failures, the above basic read-one-write-all algorithm can be extended. A read operation is allowed if at least one copy is available; the nearest available copy is read. A write operation, however, is allowed only if the majority of copies are available, and all available copies must be updated. When a site recovers from a failure or is reconnected, the protocol restores each replicated data item at the site by copying up-to-date values from the nearest available sites before

new requests are processed. Abbadi et al. (1985) described such an extension that tolerates a large class of failures, including processor and communication link crashes, partitioning of the communication network, lost messages and slow responding processors, and communication links.

In our transaction processing model (Figure 1), replica control is implemented by the GDM (for global transactions) and servers (for replica-update transactions). Because both the GDM and servers are new modules added at integration time, replica control protocols can be easily implemented, i.e., replica control in MDBSs is essentially identical to that in traditional database systems. Throughout this article, we assume that a modified version of the above mentioned read-one-write-all-available protocol is implemented in our transaction processing model.

3.3 Concurrency Control

The most commonly used concurrency control protocol in traditional database environments is two-phase locking. Many other protocols have also been proposed (Bernstein et al., 1987). None of these, however, can be applied directly to MDBSs, because transactions in that environment are managed by a set of autonomous local transaction managers. The GTM, responsible for global consistency, is superimposed on those local transaction managers and therefore has neither information about nor direct control over local executions.

Given these constraints, the GTM may adopt one of two approaches. The *pessimistic approach* prevents inconsistencies by imposing restrictions on global transactions and their executions, while the *optimistic approach* detects and resolves inconsistencies afterwards by aborting global transactions. Although the latter method may provide a higher degree of concurrency, its tendency to abort replica-update transactions (and therefore violate availability requirements) renders it inappropriate for use in a replicated MDBS.

Several pessimistic protocols have been proposed to ensure consistency for non-replicated MDBSs (Alonso et al., 1987; Breitbart and Silberschatz, 1988; Du et al., 1991). These protocols, as well as other optimistic protocols proposed in the literature (Georgakopoulos et al., 1991), are all centralized, in that all global transactions accessing multiple sites are controlled by a central agent, the GTM. Therefore, they cannot be applied directly to replicated MDBSs, because replica-update transactions access multiple sites but are not under control of the GTM.

In a simple solution to the above problem, the replica-update transaction is controlled by the GTM through one of two methods. Replica-update transactions may be submitted to the GTM, or the local servers may consult the GTM before submitting the replica-update transactions to the underlying DBMSs. However, these straightforward approaches present several drawbacks. Both methods engender delays in replica-update transactions. For example, let us suppose that a user has a checking account in a San Francisco bank which is replicated at the bank's New York office. The MDBS integrating the two local databases is located in New York. When cashing a check in San Francisco, the user must wait while the request from the local

server at San Francisco is sent to the GTM in New York, and a response is returned. Because the GTM is responsible for the update propagation of all replicated data, communication bottlenecks are also likely. An even greater concern is that local transactions are unable to update replicated data when the central agent is down, even if the local database system is still running. This significantly violates the requirements of local autonomy and reliability.

In summary, we have viewed replicated data management as two separate but interrelated tasks of replica and concurrency control. Because replica control in the MDBS environment remains largely unchanged from traditional models, we will focus on global concurrency control in extending the traditional strategies to support consistent access in replicated MDBSs. More specifically, we will describe a decentralized and pessimistic concurrency control protocol that overcomes the problems discussed above. Before we can do that, we need to extend the basic quasi-serializability to handle replication. The concurrency control protocol based on 1QSR will be presented in Section 5.

4. One-Copy Quasi-Serializability

First we will review the basic quasi-serializability theory. Then we will generalize the theory to deal with replication, and lead to the formulation of 1QSR.

4.1 An Overview of Quasi-Serializability

Quasi-serializability is a correctness criterion for global concurrency control in MDBSs. It is weaker than serializability and therefore is easier to ensure. There are two reasons for using quasi-serializability. First, due to the demands of local autonomy, it is difficult to maintain serializability in MDBSs. We have already pointed out that the only control the GTM has over local executions is in the submission of global subtransactions. It is well known that a transaction following another in an execution may effectively precede the latter in serialization order (Papadimitriou, 1986). The second reason is the independence of local transactions, which are designed and executed independently. There is no direct precedence between local transactions at different sites. Although global transactions may introduce indirect precedence between two local transactions (e.g., local transaction L_1 precedes global transaction G_1 at site S_1, which in turn precedes another local transaction, L_2 at site S_2), this precedence does not necessarily imply interaction between local transactions. Indeed, such precedence can often be prevented by delaying global transactions.

Informally, a global execution is quasi-serializable if it is equivalent to a quasi-serial execution in which global transactions are executed sequentially. The basic quasi-serializability theory assumes that there are no data replicated at the global level and therefore no replica-update transactions.

Definition 4.1. (Quasi-serial executions) A global execution, $E = \{E_1, E_2, ..., E_n\}$, is quasi-serial if

1. each local execution E_i is serializable; and
2. there is a total ordering (called *quasi-serialization order*) over $\mathcal{GT}$ so that if T_i precedes T_j in the order, all operations of T_i are executed before those of T_j in each local execution.

Definition 4.2. (Quasi-serializable executions) A global execution is quasi-serializable if it is equivalent to a quasi-serial execution of the same set of transactions.

Example 4.1. Consider a non-replicated MDBS, $\langle\{S_1, S_2\}, \{G_1, G_2, L_1, L_2\}\rangle$, where $S_1 = \langle\mathcal{D}_1, \mathcal{P}_1\rangle$ and $S_2 = \langle\mathcal{D}_2, \mathcal{P}_2\rangle$. $\mathcal{P}_1 = \mathcal{D}_1 = \{a, b\}$ and $\mathcal{P}_2 = \mathcal{D}_2 = \{c, d\}$. $G_1, G_2 \in \mathcal{GT}$ are global transactions and $L_1 \in \mathcal{LT}_1$ and $L_2 \in \mathcal{LT}_2$ are local transactions at sites S_1 and S_2, respectively.

$G_1 = \{G_{1,1}, G_{1,2}\}$, where $G_{1,1} : w_{g_1}(a)$ and $G_{1,2} : r_{g_1}(c)$.
$G_2 = \{G_{2,1}, G_{2,2}\}$, where $G_{2,1} : r_{g_2}(b)$ and $G_{2,2} : w_{g_2}(d)$.
$L_1 : r_{l_1}(a)w_{l_1}(b)$.
$L_2 : w_{l_2}(c)r_{l_2}(d)$.
Let $E = \{E_1, E_2\}$ be a global execution of G_1, G_2, L_1 and L_2, where
$E_1 : w_{g_1}(a)r_{l_1}(a)w_{l_1}(b)r_{g_2}(b)$, and
$E_2 : w_{l_2}(c)r_{g_1}(c)w_{g_2}(d)r_{l_2}(d)$.

By definition, E is quasi-serializable. However, it is not serializable. □

In quasi-serializable executions, global transactions affect each other in a partial order because they are executed sequentially in the equivalent quasi-serial executions. For example, G_1 affects G_2 (Example 4.1), but not vice versa. Global and local transactions accessing the same site also affect each other in a partial order because of local serializability. It is possible, however, for two local transactions at different sites to affect each other bilaterally. L_2 affects L_1 if $r_{l_1}(a)$ reads indirectly from $w_{l_2}(c)$ (i.e., $w_{g_1}(a)$ depends on $r_{g_1}(c)$). Similarly, L_1 affects L_2 if $w_{g_2}(d)$ depends on $r_{g_2}(b)$. Generally, a local transaction at site S_1 will be affected by another local transaction at site S_2 only if there is a global transaction whose subtransaction at site S_1 depends on the subtransaction at site S_2. Therefore, the interaction between local transactions at different sites can be prevented by controlling remote information flow in global transactions. L_2 will not affect L_1 if G_2 is delayed until either L_1 or L_2 is concluded.

Quasi-serializability proves easier to ensure than serializability, because the order of global transactions is more independent of local transactions in the former than in the latter. Some local transactions may mandate serialization orders, but such constraints are not made on quasi-serialization orders. For example, G_1 must follow G_2 in serialization order because of local transaction L_1. This is, however, not the case in quasi-serialization order.

4.2 One-Copy Quasi-Serializable Executions

There are two issues we must consider when extending quasi-serializability to deal with replication. First, if a transaction reads a data item from another transaction at the logical level, it should read the corresponding copy from the transaction. Second, since both global and replica-update transactions may update replicated data, their execution must be coordinated in a consistent way.

Definition 4.3. (One-copy quasi-serial executions) A global execution, $E = \{E_1, E_2, ..., E_n\}$, of a set of well-formed transactions is one-copy quasi-serial if

1. each local execution E_i is serializable;
2. there is a total ordering (called *quasi-serialization order*) over $\mathcal{GT} \cup \mathcal{RT}$, such that if T_i precedes T_j in the order, all operations of T_i are executed before those of T_j in each local execution; and
3. for $T_i, T_j \in \mathcal{GT} \cup \mathcal{RT}$, if T_i reads d_1^s (a copy of d_1 at site S_s) and T_j is the last transaction before T_i in the quasi-serialization order that writes d_1, then T_j writes d_1^s.

The first two conditions of one-copy quasi-serial executions are the same as those of basic quasi-serial executions (if we consider replica-update transactions as global transactions). The last condition guarantees that, although transactions may access different copies of a replicated data item, the execution is equivalent to an execution on a single copy database. Therefore, one-copy quasi-serial executions define, from a user's single-copy point of view, correct (or acceptable) executions.

Example 4.2. Consider a replicated MDBS, $\langle\{S_1, S_2\}, \{G_1, G_2, R_1, R_2, L_1, L_2\}\rangle$, where $S_1 = \langle\mathcal{D}_1, \mathcal{P}_1\rangle$ and $S_2 = \langle\mathcal{D}_2, \mathcal{P}_2\rangle$. $\mathcal{D}_1 = \{d_1\}, \mathcal{P}_1 = \{d_1^1, d_2^1\}$, $\mathcal{D}_2 = \{d_2, d_3\}$. $\mathcal{P}_2 = \{d_1^2, d_2^2, d_3^2\}$. $G_1, G_2 \in \mathcal{GT}$ are global transactions; $R_1 \in \mathcal{RT}_2$ and $R_2 \in \mathcal{RT}_1$ are replica-update transactions; $L_1 \in \mathcal{LT}_1$ and $L_2 \in \mathcal{LT}_2$ are local transactions.

$G_1 = \{G_{1,1}, G_{1,2}\}$, where $G_{1,1} : w_{g_1}(d_1^1)$ and $G_{1,2} : w_{g_1}(d_1^2)$.
$G_2 = \{G_{2,1}, G_{2,2}\}$, where $G_{2,1} : r_{g_2}(d_1^1)w_{g_2}(d_2^1)$ and $G_{2,2} : w_{g_2}(d_2^2)$.
$R_1 = \{R_{1,1}, R_{1,2}\}$, where $R_{1,1} : w_{r_1}(d_1^1)$ and $R_{1,2} : r_{r_1}(d_2^2)r_{r_1}(d_3^2)$.
$R_2 = \{R_{2,1}, R_{2,2}\}$, where $R_{2,1} : w_{r_2}(d_2^1)$ and $R_{2,2} : w_{r_2}(d_2^2)$.
$L_1 : r_{l_1}(a_1^1)r_{l_1}(d_2^1)$.
$L_2 : w_{l_2}(d_3^2)r_{l_2}(d_2^2)$.

Let $E = \{E_1, E_2\}$ be a global execution of G_1, G_2, R_1, R_2, L_1 and L_2, where

$E_1 : r_{l_1}(d_1^1)w_{g_1}(d_1^1)w_{r_1}(d_1^1)r_{g_2}(d_1^1)w_{g_2}(d_2^1)w_{r_2}(d_2^1)r_{l_1}(d_2^1)$, and
$E_2 : w_{g_1}(d_1^2)r_{r_1}(d_2^2)r_{r_1}(d_3^2)w_{l_2}(d_3^2)r_{l_2}(d_2^2)w_{g_2}(d_2^2)w_{r_2}(d_2^2)$.

E is one-copy quasi-serial, and the quasi-serialization order is $G_1 \rightarrow R_1 \rightarrow G_2 \rightarrow R_2$. Note that G_2 reads d_1^1 from R_1, which is the last transaction before G_2 in the quasi-serialization order that writes d_1. If, instead of reading d_1^1, G_2 reads d_1^2, the execution will not be one-copy quasi-serial, because G_2 reads d_1 from G_1. We observe that E is not one-copy serializable. □

Definition 4.4. (One-copy quasi-serializable executions) A global execution of a set of well-formed transactions is one-copy quasi-serializable if it is equivalent to a one-copy quasi-serial execution of the same set of transactions.

Example 4.3. For the same MDBS in Example 4.2, let $E' = \{E_1', E_2'\}$, where

$$E_1' : w_{g_2}(d_2^1) w_{r_2}(d_2^1) r_{l_1}(d_1^1) r_{l_1}(d_2^1) w_{g_1}(d_1^1) w_{r_1}(d_1^1) r_{g_2}(d_1^1)$$
$$E_2' : w_{g_1}(d_1^2) r_{r_1}(d_2^2) r_{r_1}(d_3^2) w_{l_2}(d_3^2) r_{l_2}(d_2^2) w_{g_2}(d_2^2) w_{r_2}(d_2^2)$$

E' is one-copy quasi-serializable, because it is equivalent to E of Example 4.2. □

The correctness of one-copy quasi-serializable executions is derived directly from that of quasi-serializable executions by considering replica-update transactions as global transactions. Local transactions at different sites do not affect each other directly, because they are not allowed to update replicated data. Global and replica-update transactions, which update replicated data, affect each other in a partial order, because they are executed sequentially in the one-copy quasi-serial execution.

5. A Decentralized Concurrency Control Protocol

As we have seen, decentralized concurrency control in replicated MDBSs is a difficult problem with no general solution. Because a replica-update subtransaction is executed and committed independently at its host site, it is inevitable that two conflicting replica-update and/or global transactions may be scheduled inconsistently by two servers. To address the problem, we distinguish between resolvable and unresolvable conflicts. As their names suggest, resolvable conflicts can be resolved by local servers at run time, while unresolvable conflicts cannot. In this section, we first present the basic protocol. We then identify resolvable conflicts in a replicated MDBS and show how they can be resolved at run time. We also show how unresolvable conflicts can be prevented by imposing restrictions on the accessibility of global and replica-update transactions. A concurrency control algorithm based on the protocol is also presented.

5.1 Basic Protocol

To maintain 1QSR of global executions, it is sufficient to execute global and replica-update transactions in a consistent order at all local sites. The key to achieving this

goal is for each server to have the global execution order before it actually executes the transactions. In conventional pessimistic protocols (e.g., altruistic locking, Alonso et al., 1987; access graph, Du et al., 1991), the GTM is responsible for generating and sending the order to the servers. The servers enforce the order by delaying any transactions that arrived too early.

This approach is not applicable to replicated MDBSs, because the GTM is not aware of replica-update transactions at local sites. In fact, there is no central agent that possesses knowledge of all global and replica-update transactions, and therefore it is impossible to construct such an order statically (i.e., before transactions are executed).

In the proposed protocol, the GTM generates the order for global transactions only. It is each server's responsibility to construct its own order for both global and replica-update transactions at the site. Primary copies are accessed in this order, and the order is also propagated to other sites. Restrictions, however, must be imposed on the transactions that update replicated data, so that if they are ordered consistently at different sites, at least the inconsistencies can be resolved at run time.

The following concepts will be useful in describing the procedure of generating the consistent order. A *global order* is a linear order over all global transactions, and a *global total order* is a linear order over all global and replica-update transactions. A *local order* at a site is a linear order over all global and replica-update transactions originating at the site, and a *local total order* is the projection of a global total order of all transactions executed at the site.

We say that two orders O_1 and O_2 are *compatible* if for any transactions T_1 and T_2 that appear in both orders, either $T_1 \stackrel{O_1}{\rightarrow} T_2$ and $T_1 \stackrel{O_2}{\rightarrow} T_2$, or $T_1 \stackrel{O_1}{\leftarrow} T_2$ and $T_1 \stackrel{O_2}{\leftarrow} T_2$. Given a set of compatible local orders, there exists at least one global total order that is compatible with all local orders.

Example 5.1. Let O_1 and O_2 be two local orders at site S_1 and S_2, respectively.

$O_1 : G_1 \rightarrow R_2 \rightarrow R_3 \rightarrow G_4 \rightarrow R_5 \rightarrow G_6 \rightarrow R_7 \rightarrow R_8 \rightarrow G_9$.

$O_2 : G_1 \rightarrow R_{10} \rightarrow G_4 \rightarrow R_{11} \rightarrow R_{12} \rightarrow G_6 \rightarrow R_{13} \rightarrow G_9$.

Where $G_1, G_4, G_6, G_9 \in \mathcal{GT}$, $R_2, R_4, R_5, R_7, R_8 \in \mathcal{RT}_1$, and $R_{10}, R_{11}, R_{12}, R_{13} \in \mathcal{RT}_2$.

O_1 and O_2 are clearly compatible, and there exists a global total order O that is compatible with both O_1 and O_2.

$O : G_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_{10} \rightarrow G_4 \rightarrow R_5 \rightarrow R_{12} \rightarrow G_6 \rightarrow R_7 \rightarrow R_8 \rightarrow R_{13} \rightarrow G_9$.

O_1 and O_2 are also compatible with the following local total order $\hat{O}$.

$\hat{O} : G_1 \rightarrow R_{10} \rightarrow R_2 \rightarrow R_3 \rightarrow G_4 \rightarrow R_{12} \rightarrow R_5 \rightarrow G_6 \rightarrow R_{13} \rightarrow R_7 \rightarrow R_8 \rightarrow G_9$. □

In the proposed protocol, local total orders are constructed in the following four steps:

1. The GTM determines the global order and submits the order along with each global subtransaction to all local sites.

2. After receiving the global order, a server constructs a local total order by adding all replica-update/propagation subtransactions executed or being executed at the site. The order is compatible with the global order and reflects the actual execution (and therefore quasi-serialization) order at the site.

3. For each locally executed replica-update subtransaction, the server sends both the replica-propagation subtransaction and the local total order to relevant sites.

4. After receiving local total orders from other sites, a server compares it with its own order. Inconsistencies (if any) are resolved at this step.

Local orders at different sites are always compatible, because the replica-update transaction sets at different sites are disjoint. Therefore, there is always a global total order. Because local servers construct local total orders independently, it is possible that they construct incompatible orders. The question is whether and under what condition the inconsistency can be resolved without aborting transactions.

5.2 Resolvable Conflicts

Although it is impossible to prevent all inconsistencies, it is possible to resolve some inconsistencies at run time, as illustrated by the following example.

Example 5.2. Consider a fully replicated MDBS, $\langle\{S_1, S_2\}, \{G_1, R_2, L_3\}\rangle$, where $S_1 = \langle\mathcal{D}_1, \mathcal{P}_1\rangle$ and $S_2 = \langle\mathcal{D}_2, \mathcal{P}_2\rangle$. There are two data items belonging to S_1 which are replicated at both sites: $\mathcal{D}_1 = \{d_1, d_2\}, \mathcal{D}_2 = \emptyset, \mathcal{P}_1 = \{d_1^1, d_2^1\}$, and $\mathcal{P}_2 = \{d_1^2, d_2^2\}$. $G_1 \in \mathcal{GT}$ is a global transaction, $R_2 \in \mathcal{RT}_1$ are replica-update transactions, and $L_3 \in \mathcal{LT}_2$ is a local transaction, defined as follows.

$G_1 = \{G_1^1, G_1^2\}$, where $G_1^1 : r_{g_1}(d_1^1)w_{g_1}(d_1^1)$ and $G_1^2 : w_{g_1}(d_1^2)$.
$R_2 = \{R_2^1, R_2^2\}$, where $R_2^1 : r_{r_2}(d_1^1)w_{r_2}(d_1^1)r_{r_2}(d_2^1)w_{r_2}(d_2^1)$ and
$R_2^2 : w_{r_2}(d_1^2)w_{r_2}(d_2^2)$.
$L_3 : r_{l_3}(d_1^2)r_l(d_2^2)$.

Suppose that G_1 and R_2 were submitted to the GTM and to the server at S_1, respectively, at about the same time. Due to the communication delay, R_2^1 was already committed before G_1^1 arrived at S_1, but R_2^2 arrived at S_2 after G_1^2 was committed. In other words, the local total orders constructed at two sites are $R_2 \rightarrow G_1$ and $G_1 \rightarrow R_2$, respectively.

Let $E = \{E_1, E_2\}$ be the execution of G_1 and R_2, where
$E_1 : r_{r_2}(d_1^1)w_{r_2}(d_1^1)r_{r_2}(d_2^1)w_{r_2}(d_2^1)r_{g_1}(d_1^1)w_{g_1}(d_1^1)$, and
$E_2 : r_{l_3}(d_1^2)w_{g_1}(d_1^2)w_{r_2}(d_1^2)w_{r_2}(d_2^2)r_{l_3}(d_2^2)$.

The execution is not one-copy quasi-serializable, because two local total orders are not compatible with each other. The inconsistency cannot be prevented by the GTM, because it is not aware of R_2. The server at S_1 has no information about E_2, therefore it is unaware of the inconsistency and will commit R_2^1 as soon as it finishes. The server at S_2 detects the inconsistency after R_2^2 arrives, too late to prevent this from occurring, because R_2^1 and G_1^2 have already been committed. □

The inconsistency in the above example is caused by the obsolete write $w_{r_2}(d_1^2)$ of R_2. It is obsolete because it conflicts with the write operation $w_{g_1}(d_1^2)$ of G_1 and arrived at S_2 too late (i.e., after G_1^2 completes). The inconsistency can be prevented by servers at S_1 and S_2 if both G_1 and R_2 follow two-phase commit protocol, as in conventional distributed database systems. This is, however, undesirable for two simple reasons: we do not want to delay the commitment of R_2^1 until R_2^2 completes, and we do not want to abort G_1 or R_2 unless it is really unavoidable.

In this particular example, the inconsistency can be resolved at S_2 without aborting any transactions: by simply ignoring the obsolete write (i.e., $w_{r_2}(d_1^2)$).

Let $\hat{E} = \{\hat{E}_1, \hat{E}_2\}$ be the resultng execution, where
$\hat{E}_1 : r_{r_2}(d_1^1)w_{r_2}(d_1^1)r_{r_2}(d_2^1)w_{r_2}(d_2^1)r_{g_1}(d_1^1)w_{g_1}(d_1^1)$, and
$\hat{E}_2 : r_{l_3}(d_1^2)w_{g_1}(d_1^2)w_{r_2}(d_2^2)r_{l_3}(d_2^2)$.

Clearly, $\hat{E}$ is semantically equivalent to a one-copy quasi-serializable execution $\bar{E} = \{\bar{E}_1, \bar{E}_2\}$, where
$\bar{E}_1 : r_{r_2}(d_1^1)w_{r_2}(d_1^1)r_{r_2}(d_2^1)w_{r_2}(d_2^1)r_{g_1}(d_1^1)w_{g_1}(d_1^1)$, and
$\bar{E}_2 : r_{l_3}(d_1^2)w_{r_2}(d_1^2)w_{r_2}(d_2^2)w_{g_1}(d_1^2)r_{l_3}(d_2^2)$.

Unfortunately, not all inconsistencies lend themselves to such a resolution. In Example 5.2, if G_1 not only updates d_1^2 but also reads d_2^2, simply ignoring R_2^2 will result in a semantically different execution. Therefore, it is important to distinguish between resolvable and unresolvable conflicts. In order to do so, we need to formalize the notion of semantic equivalence. The following concept of virtual equivalence is an extension of the traditional notion of execution equivalence (Bernstein et al., 1987).

Given two executions, E and $\hat{E}$, over transaction sets $\{T_1, T_2, ..., T_k\}$ and $\{\hat{T}_1, \hat{T}_2, ..., \hat{T}_k\}$, respectively, we say that E is *virtually equivalent* to $\hat{E}$ if

1. $\mathcal{O}(\hat{T}_i) \subseteq \mathcal{O}(T_i)$ for $i = 0, 1, ..., k$;

2. for any two transactions T_i, T_j in E (hence $\hat{T}_i, \hat{T}_j$ in $\hat{E}$) and for any data item x, if T_i reads x from T_j in E, then $\hat{T}_i$ reads x from $\hat{T}_j$ in $\hat{E}$; and

3. for each data item x, if T_i is the last transaction that writes x in E, then $\hat{T}_i$ is the last transaction that writes x in $\hat{E}$.

Notice that two virtually equivalent executions do not have to be defined on the same set of transactions. The difference is a set of obsolete operations that have no impact on the semantics of the executions. For example, $\hat{E}$ is virtually equivalent to $\bar{E}$. The difference between two executions is a dead-write $w_{r_2}(d_1^2)$.

Informally, a conflict between two transactions is resolvable if one of the conflicting operations can be removed without changing the semantics of execution.

Definition 5.1. (Resolvable conflicts) Given a local execution, E_l, defined over transaction set T_l and two conflicting operations, $o_i \in \mathcal{O}(T_i)$ and $o_j \in \mathcal{O}(T_j)(i \neq j$ and $o_i precedes o_j)$, the conflict is resolvable if there exists $\hat{E}_l$ and $\bar{E}_l$ so that

$\hat{E}_l = E/o_j$;[4]

$\bar{E}_l$ is defined over T_l, and T_i^l does not conflict with T_j^l in $\bar{E}_l$; and

$\hat{E}_l$ is virtually equivalent to $\bar{E}_l$.

According to Definition 5.1, the conflict between $w_{g_1}(d_1^2)$ and $w_{r_2}(d_1^2)$ in E of Example 5.2 is resolvable by ignoring $w_{r_2}(d_1^2)$.

We also say that a conflict between two transactions is resolvable if conflicts between their operations are all resolvable. Conflicts between two transactions that are not resolvable are called unresolvable conflicts.

5.3 Preventing Unresolvable Conflicts

In order to ensure one-copy quasi-serializability, it is sufficient for a protocol to prevent all unresolvable conflicts. Generally, there are four types of global conflicts in a replicated MDBS:

1. conflicts between two global transactions;
2. conflicts between two replica-update transactions at the same site;
3. conflicts between a global and a replica-update transaction; and
4. conflicts between two replica-update transactions at different sites.

The first two types are easily prevented in the protocol. For example, the global order is determined by the GTM and sent to all relevant local sites along with these global transactions. Local servers, therefore, can forestall Type 1 conflicts simply by

4. E/o_j means removing o_j from E.

delaying global subtransactions which arrived too early. Similarly, Type 2 conflicts can be prevented by the server, as it has full control over both transactions.

As illustrated in Example 5.2, Type 3 conflicts are unpreventable at a non-host site. It is also impossible to prevent Type 4 conflicts if two replica-update subtransactions are issued to two sites at about the same time. In this case, both replica-update subtransactions are committed before submitting the corresponding replica-propagation subtransactions, and therefore no server is aware of both transactions until the two updating subtransactions have completed. In either case, however, the conflict is caused by an obsolete replica-propagation subtransaction.

Generally, the resolvability of these two types of conflicts depends on the data accessed by the two subtransactions. Because a replica-propagation subtransaction contains writes only, a conflict is resolvable if the subtransaction does not write data previously read by the conflicting (global or replica-update) subtransactions, and as in Example 5.2, the conflicts can be resolved by ignoring the obsolete writes.

The discussion presented above can be summarized by the following theorem (see Appendix A for proof).

Theorem 5.1. The global execution generated by the protocol is virtually equivalent to a one-copy quasi-serializable execution of all original transactions, if the following two conditions hold:

1. $\forall G_0 \in \mathcal{GT}$ and $R_i \in \mathcal{RT}_i, \mathcal{R}(G_0^j) \cap \mathcal{W}(R_i^j) = \emptyset$, for all $j \neq i$; and
2. $\forall R_i \in \mathcal{RT}_i$ and $R_j \in \mathcal{RT}_j, (\mathcal{W}(R_i) \cap \mathcal{R}(R_j)) \cup (\mathcal{R}(R_i) \cap \mathcal{W}(R_j)) = \emptyset$.

Basically, the conditions set forth in the theorem ensure that the write set of a replica-propagation subtransaction does not overlap with the read set of global and replica-update/propagation subtransactions from other sites. The conditions can be satisfied by imposing the following restrictions on the accessibility of global and replica-update transactions:

1. $\forall G \in \mathcal{GT}, \mathcal{R}(G) \cap (\mathcal{P}_i \cap f(D_j)) = \emptyset$. That is, global transactions do not read non-primary copies of replicated data.
2. $\forall R \in \mathcal{RT}_i, \mathcal{O}(R) \subseteq \mathcal{P}_i \cap f(D_i)$. That is, replica-update transactions only access data that logically belong to the host site.

Given these restrictions, the first condition of the theorem holds true because a replica-propagation subtransaction updates only non-primary copies which are not readable by global transactions. The validity of the second condition arises from the disjointness of replicated data sets at different sites.

We believe that the restrictions proposed do not conflict with the requirements of local autonomy and availability, because no restriction is imposed on local transactions, and replica-update transactions can independently access all data items (regardless of replication) belonging to their host sites.

5.4 The Algorithm

The algorithm for the proposed protocol consists of two parts: a global scheduler and a set of local servers. The global scheduler assigns a unique global identification, or GID, to each global transaction. This number, incremented with each use, reflects the global order of the transaction and is submitted along with global subtransactions to servers at local sites.

Local servers at each site not only enforce the global order, but also coordinate executions of replica-update transactions. A server executes global and replica-update transactions sequentially at each site (Section 6 presents an optimization of this approach). Each replica-update transaction issued at a site is incrementally assigned an identification number (local identification, or LID) by the server. The number and its relationship to the global order (e.g., LID_0 follows GID_1 but precedes GID_2) reflect the local order of the site.

In order for local servers to construct local total order, each replica-propagation subtransaction R_i^j of R_i is augmented with the following data structures:

1. G_ID: The GID of global transactions that precede R_i in the local order.
2. R_ID: The LID of R_i.
3. $R\_LID[k]$: The largest LID of replica-update transactions at site S_k that have propagation subtransactions at both S_i and S_j and have been scheduled before R_i^i at site S_i.

With the same rationale, each global subtransaction G_1^i is augmented with the following data structure: G_ID: The GID of G_1.

The local total orders are never explicitly constructed. However, they can be derived from the values of the above data structures. Given a global subtransaction, G_1^j, and a replica-update subtransaction, R_1^j, G_1^j precedes R_1^j at site S_j if and only if $G_1^j.G\_ID \leq R_1^j.G\_ID$. Similarly, a replica-propagation subtransaction R_k^j from site S_k precedes R_i^j, a replica-propagation subtransaction from site S_i, if $R_k^j.R\_LID[k] \leq R_i^j.R\_ID$.

The following data structures are used by a server to assign G_ID, R_ID, and R_LID values to a replica-propagation subtransaction.

1. $DELAYED$: A set of global and replica-update/propagation subtransactions that are delayed due to preceding subtransactions in the local total order that have not yet completed. Note that a local transaction is never delayed.
2. $ARRIVED$: A set of global and replica-update/propagation subtransactions that are either delayed or still active.

3. $FINISHED$: A set of global subtransactions that are either committed or aborted.

4. $CURRENT\_G\_ID$: The largest GID of global subtransactions that are either active or finished at site S_i.

5. $LAST\_G\_ID$: The largest GID of global transactions that arrived at site S_i.

6. $CURRENT\_R\_ID[k]$: The largest LID of replica-update transactions that belong to S_k and are either active or finished at S_i.

7. $LAST\_R\_ID[k]$: The largest LID of replica-update transactions that belong to S_k and are arrived at S_i.

8. $CURRENT\_R\_ID[k]$: The largest LID of replica-propagation subtransactions that belong to S_k and are either active or finished at S_i.

9. $GT[k]$: The global subtransaction of GID k.

Each server consists of two components: SERVER_SUBMISSION and SERVER_TERMINATION. The former is invoked each time a transaction is received, while the latter is invoked only when a transaction has finished. SERVER_SUBMISSION invokes subprocedures SERVER_GT, SERVER_LT, and SERVER_RT to schedule global subtransactions, local replica-update transactions, and remote replica-propagation subtransactions, respectively.

Procedure SERVER_GT enforces the global order by comparing the G_ID of each global subtransaction with $CURRENT\_G\_ID$. A global subtransaction is delayed if its G_ID is greater than $CURRENT\_G\_ID$ (meaning that the previous global subtransaction has not yet completed).

Procedure SERVER_LT executes local replica-update transactions separately. Each replica-update transaction is assigned a unique R_ID, which reflects the local order in which it is executed. A replica-update transaction is delayed if the current global subtransaction has not yet completed. This not only guarantees that the replica-update transaction is executed sequentially with the global transactions, but also implies that the replica-update transaction follows the global transaction in the local order. The order is sent, along with replica-propagation subtransactions, to other sites.

Procedure SERVER_RT adopts two measures to enforce a given local total order between a global subtransaction and a replica-propagation subtransaction at a remote site. First, a replica-propagation subtransaction will be delayed if it follows a global subtransaction in the local order but arrives before the global transaction. Second, if a replica-propagation subtransaction finds that a global transaction following it in the local order has already been executed, it will simply ignore the updates that conflict with the global subtransaction (resolvable conflicts):

```
procedure SERVER_SUBMISSION_i(T)
begin
    if T is not a local transaction
       then ARRIVED = ARRIVED ∪ {T}
    case 1: /* T is a local transaction */
       submit T to the LDBS
    case 2: /* T is a global transaction */
       SERVER_GT_i(T)
    case 3: /* T is a local replica-update transaction */
       SERVER_LT_i(T)
    case 4: /* T is a replica-propagation subtransaction from site S_k */
       SERVER_RT_i(T)
endprocedure SERVER_SUBMISSION_i

procedure SERVER_TERMINATION_i(T)
begin
    ARRIVED = ARRIVED − {T}
    case 1: /* T is a local transaction */
       do nothing
    case 2: /* T is a global subtransaction */
       FINISHED = FINISHED ∪ {T}
       CURRENT_G_ID = CURRENT_G_ID + 1
    case 3: /* T is a local replica-update transaction */
       CURRENT_R_ID[i] = CURRENT_R_ID[i] + 1
    case 4: /* T is a replica-propagation subtransaction from S_k */
       CURRENT_R_ID[k] = CURRENT_R_ID[k] + 1
    for each T ∈ DELAYED do
       DELAYED = DELAYED − {T}
       SERVER_SUBMISSION_i(T)
    endfor
endprocedure SERVER_TERMINATION_i(T)

procedure SERVER_GT_i(G_i)
/* G_i is a global subtransaction */
begin
    LAST_G_ID = max(LAST_G_ID, G_i.G_ID)
    if (G_i.G_ID⟩CURRENT_G_ID)
       or (G_i.G_ID = CURRENT_G_ID)
       and (∃T ∈ ARRIVED such that T.G_ID⟨G_i.G_ID)
       then DELAYED = DELAYED ∪ {G_i}
       else if G_i.G_ID = CURRENT_G_ID
          then submit G to the LDBS
          else return(ERROR)
          endif
```

```
        endif
endprocedure SERVER_GT_i

procedure SERVER_LT_i(R_i)
/* R_i is a local replica-update transaction */
begin
    for each subtransaction R_i^j of R_i do
      if (i ≠ j) /* replica-propagation subtransaction */
        then R_i^j.R_ID = LAST_R_ID
          LAST.R_ID = LAST_R_ID + 1
          for k = 1, 2, ..., n do
            R_i^j.R_LID[k] = LAST_R_ID[k]
          endfor
          R_i^j.G_ID = LAST_G_ID
          send R_i^j to site S_j
        else if GT[CURRENT_G_ID] ∈ FINISHED
          and R_i^i.R_ID⟩CURRENT_R_ID
          then submit R_i^i to the LDBS
          else DELAYED = DELAYED ∪ {R_i^i}
          endif
        endif
      endfor
endprocedure SERVER_LT_i

procedure SERVER_RT_i(R_k^i)
/* R_k^i is a replica-propagation subtransaction from site D_k */
begin
    LAST_R_ID[k] = max(LAST_R_ID[k], R_k.R_ID)
    if (R_k^i.G_ID⟩CURRENT_G_ID) or (R_k^i.R_ID⟩CURRENT_R_ID[k])
      then DELAYED = DELAYED ∪ {R_k^i}
      endif
    if ∃G ∈ FINISHED such that G.G_ID⟩R_k^i.G_ID
      then for each w_k ∈ W(R_k^i) ∩ W(G) do
          remove w_k from R_k^i
        endfor
      endif
    submit R_k^i to the LDBS
endprocedure SERVER_RT_i
```

6. Related Issues

The previous section presented a decentralized concurrency control protocol capable of coordinating execution orders of global and replica-update transactions. In this section, issues related to the implementation and enhancement of this protocol will be discussed.

6.1 Improving Performance

The performance of the protocol described in the previous section may be improved in several ways. Here we suggest an approach to improving concurrency among global transactions.

The algorithm executes global transactions sequentially by assigning a unique number to each global transaction. As the following example illustrates, global transactions need not be sequentially executed to ensure 1QSR.

Example 6.1. Consider an MDBS, $\langle \{S_1, S_2, S_3\}, \{G_1, G_2\} \rangle$, where $S_i = \langle D_i, P_i \rangle$. Let $D_1 = \{a_1\}, D_2 = \{b_1\}, D_3 = \{c_1\}, P_1 = \{a_1^1, b_1^1\}, P_2 = b_1^2, a_1^2\}$ and $P_3 = \{c_1^3, b_1^3\}$.

$G_1 = \{G_1^1, G_1^2\}$, where $G_1^1 : w_{g_1}(a_1^1)$ and $G_1^2 : w_{g_1}(a_1^2)$.
$G_2 = \{G_2^2, G_2^3\}$, where $G_2^2 : w_{g_2}(b_1^2)$ and $G_2^3 : w_{g_2}(b_1^3)$.

Regardless of the manner in which G_1 and G_2 are scheduled at three sites, the global execution is always one-copy quasi-serializable. This holds true even in presence of any local transactions. □

In general, the quasi-serialization order of a set of global transactions at a local site is significant to the quasi-serializability of the global execution only if the global transactions form a cyclic access graph (Du et al., 1991). The access graph of a set of global transactions is defined as the union of the access graphs of each global transaction, which in turn are defined as the acyclic graphs of sites accessed by the transaction. For example, the access graph of $\{G_1, G_2\}$ in the above example is $S_1 - S_2 - S_3$, which is acyclic.

To take advantage of this property of global executions, the global scheduler needs to maintain the access graphs for each active global transaction. Global transactions that form an acyclic access graph can be executed concurrently at local sites and therefore are assigned the same GID number. As far as concurrency control is concerned, servers do not need to distinguish among global subtransactions that share the same GID number.

6.2 Commitment of Replica-Update Transactions

Transaction atomicity in distributed database systems is generally ensured by a two-phase commit protocol (Gray, 1978). The protocol requires that each LDBS support a prepared state for transactions. A transaction is in the prepared state if

it can be neither unilaterally aborted nor committed by the LDBSs. Such a solution may be precluded in MDBSs by the inability of LDBSs to support a prepared state. This difficulty can be addressed by redoing writes of globally committed but locally aborted subtransactions (Wolski and Veijalainen, 1990). Special attention must be paid, because LDBSs treat the original subtransaction and redo subtransactions separately, and they may interleave improperly with local transactions. Mehrontra et al. (1991) discussed the problem by presenting sufficient conditions under which the inconsistent interleaves can be avoided.

Under the restrictions outlined in the previous section, replica-update transactions need not follow two-phase commit protocol because they access only data that logically belongs to their host site and update only non-primary copies at remote sites.

Consider a replica-update transaction, $R_i \in RT_i$, that consists of two subtransactions, R_i^i and R_i^j where $j \neq i$. The only purpose of R_i^j is to propagate updates of R_i^i to site S_j. The server at site S_i can commit R_i^i as soon as it successfully completes and before sending R_i^j to site S_j. This is possible because R_i^i does not depend on the execution of R_i^j, and R_i^j does not conflict with local and replica-update transactions at site S_j (i.e., they do not update common data). Therefore, R_i^j can be executed and re-executed until it succeeds. The only transactions that write-conflict with R_i^j are global transactions and replica-update transactions from the same site, and the correct execution order is guaranteed by the concurrency control protocol.

Note that the above observation would not hold true if replica-update transactions were allowed to read data at remote sites. In such an instance, R_i^i cannot be safely committed until the successful completion of R_i^j, because re-execution of R_i^j may result in the re-execution of R_i^i.

6.3 Related Work

The approach presented in this article belongs to the class of protocols that *update local copies as soon as possible* without waiting for the completion of update propagations. We now compare the proposed protocol with another protocol of the class, one proposed for Tandem's EMPACT^{TM} system.

EMPACT^{TM} is a distributed database application system for manufacturing information control (Norman and Anderton, 1983). There are two types of data in the system: replicated global data and non-replicated local data. Requests to access global and local data can be issued at all local sites. Local copies are updated immediately by local servers, while updates to replicated data are also propagated to remote sites.

While both protocols provide prompt update of local copies, they differ in several respects. First, the system models are different; EMPACT^{TM} is a traditional distributed database and makes no distinction among global, local, and replica-update

transactions. A transaction can read/write all data, both global and local at the site where it is issued. Transactions are managed by the TMF (transaction and recovery manager), which is part of the system. In our model, transactions are managed by the GTM and by a set of autonomous LTMs. Second, the objective of the EMPACT^{TM} protocol is site autonomy and rapid response time. It allows a transaction to update local copies immediately but does not ensure the consistency (i.e., serializability) of the execution. In contrast, the proposed protocol supports both prompt and consistent updates to replicated data.

7. Conclusion

Replication is an important issue in both traditional database systems and multidatabase systems. Concurrency control, however, is more difficult in the latter. Due to the constraints of local autonomy, there can be no central agent that controls all transactions accessing replicated data. In this article, we presented a concurrency control protocol that ensures quasi-serializability. The protocol is decentralized in the sense that local servers coordinate with each other to prevent and resolve inconsistent access to replicated data by delaying subtransactions that arrive too early or by ignoring obsolete propagations that arrive too late.

It is generally impossible to guarantee a globally consistent execution order of both global and local applications. Also, it is not always possible to resolve inconsistencies. These obstacles have been addressed in this article by imposing restrictions to accessibility on the global applications and those local applications that update replicated data. Two types of inconsistencies have been distinguished: those that are resolvable by local servers and those that are not. Conditions are then established to prevent the occurrence of unresolvable inconsistencies. These restrictions are both strict enough to prevent all unresolvable inconsistencies and acceptable with respect to local autonomy and availability requirements.

An important issue that has not been addressed in this article is the extension of the protocol to survive failures. Much work in this area has been done in traditional database environments (Bernstein, 1987; Davidson, 1985). Due to the demands of local autonomy, these techniques may not directly transfer to multidatabase systems; this question merits further investigation.

Acknowledgments

The authors would like to thank Prof. Andreas Reuter and anonymous reviewers for their constructive comments. The work of the second author was supported by the Indiana Corporation for Science and Technology (CST), a PYI Award from the NSF under grant IRI-8857952, grants from the AT&T Foundation, Mobil Oil, Tektronix, UniSQL, a David Ross Fellowship from the Purdue Research Foundation, and a grant from the Software Engineering Research Center at Purdue University,

a National Science Foundation Industry/University Cooperative Research Center (NSF Grant No. ECD-8913133).

References

Abbadi, A., Skeen, D., and Cristian, F. Principles of Database Systems, Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium, Portland, OR 1985.

Alonso, R., Garcia-Molina, H., and Salem, K. Concurrency control and recovery for global procedures in federated database systems. *IEEE Data Engineering Bulletin,* 10(3):5–11, 1987.

Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Databases Systems.* Reading, MA: Addison-Wesley Publishing Co., 1987.

Breitbart, Y. and Silberschatz, A. Multidatabase update issues. *Proceedings of the International Conference on Management of Data,* Chicago, IL, 1988.

Davidson, S., Garcia-Molina, H., and Skeen, D. Consistency in partitioned networks. *ACM Computing Surveys,* 17(3):341–369, 1985.

Du, W. and Elmagarmid, A. Quasi serializability: A correctness criterion for global concurrency control in InterBase. *Proceedings of the International Conference on Very Large Data Bases,* Amsterdam, 1989.

Du, W., Elmagarmid, A., and Kim, W. Maintaining quasi serializability in HDDBSs. *Proceedings of the International Conference on Data Engineering,* Kobe, Japan, 1991.

Du, W., Elmagarmid, A., Leu, Y., and Ostermann, S. Effects of autonomy on global concurrency control in heterogeneous distributed database systems. *Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering,* Gaithersburg, MD, 1989.

Georgakopoulos, D., Rusinkiewicz, M., and Sheth, A. Transaction management in distributed heterogeneous database management systems. *Proceedings of the Seventh International Conference on Data Engineering,* Kobe, Japan, 1991.

Gray, J. Notes on database operating systems. In: *Lecture Notes in Computer Science, Operating System: An Advanced Course,* Vol. 60, Berlin: Springer-Verlag, 1978, pp. 393–481.

Mehrontra, S., Rastogi, R., Breitbart, Y., Korth, H., and Silberschatz, A. Ensuring transaction atomicity in multidatabase systems. *Proceedings of the Eleventh ACM Symposium on the Principles of Database Systems,* San Diego, CA, 1991.

Norman, A. and Anderton, M. EMPACT: A distributed database application. *Proceedings of the National Computer Conference (AFIPS),* 1983.

Papadimitriou, C. *The Theory of Database Concurrency Control.* Montvale, NJ: AFIPS Press, 1986.

Stonebreaker, M. *Readings in Database Systems,* Palo Alto, CA: Morgan Kaufmann, 1988.

Wolski, A. and Veijalainen, J. 2PC agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase. *Proceedings of PARBASE-90,* Miami Beach, FL, 1990.

Appendix A. Proof of Theorem 5.1

Let $R_i \in \mathcal{RT}_i$ and $T_j \in \mathcal{GT} \cup (\cup_{j \neq i} \mathcal{RT}_j)$. It is sufficient to show that any inconsistency between R_i and T_j is either preventable or resolvable.

The proof consists of the following three sections.

1. T_j precedes R_i at site S_i, but follows R_i at other sites. Because T_j^i arrived at site S_i before R_i was submitted, the order (i.e., $T_j \rightarrow R_i$) is included in the local total order at S_i. The server at another site (e.g., S_k) is informed of this order by the arrival of R_i^k and will enforce it; for example, it will delay R_i^k until T_j^k has finished.

2. T_j follows R_i at site S_i, but precedes R_i at other sites and $T_j \in \mathcal{GT}$. Suppose that R_i precedes T_j at S_i, but follows it at S_k, where $k \neq i$. There are two subcases:

   2.a T_j^k directly conflicts with R_i^k at S_k. Because $(\mathcal{R}(T_j^k) \cap \mathcal{W}(R_i^k)) \cup (\mathcal{W}(T_j^k) \cap \mathcal{R}(R_i^k)) = \emptyset$, there exist $w_j(d_0^k) \in \mathcal{W}(T_j^k)$ and $w_i(d_0^k) \in \mathcal{W}(R_i^k)$ so that $w_j(d_0^k)$ precedes and conflicts with $w_i(d_0^k)$.

   E_k : .....$w_j(d_0^k) w_i(d_0^k)$.....

   Because the server at S_k knows (from the local total order carried by R_i^k) that T_j precedes R_i at S_i, it can detect the inconsistency before submitting $w_i(d_0^k)$ and can therefore resolve it by ignoring the operation. The resulting execution is virtually equivalent to the following execution:

   $\hat{E}_k$: .....$w_i(d_0^k) w_j(d_0^k)$.....

   2.b T_j^k indirectly conflicts with R_i^k at S_k. Because $\mathcal{R}(R_i^k) = \emptyset$ (see Section 2), there exist $o_j \in \mathcal{O}(T_j^k)$ and $w_i(d_0^k) \in \mathcal{W}(R_{ik})$ so that o_j precedes and indirectly conflicts with $w_i(d_0^k)$. Therefore there exist $o_1, o_2, ..., o_p$ so that $o_j \rightarrow o_1 \rightarrow o_2 \rightarrow ... \rightarrow o_p \rightarrow w_i(d_0^k)$.

   There are three types of operations that may precede and conflict with $w_i(d_0^k)$:

   - $w_g(d_0^k) \in \mathcal{W}(G)$, where $G \in \mathcal{GT}$;
   - $w_r(d_0^k) \in \mathcal{W}(R)$, where $R \in \mathcal{RT}_i$; and
   - $r_l(d_0^k) \in \mathcal{R}(L)$, where $L \in \mathcal{LT}_k$.

   Suppose that $o_p = w_g(d_0^k)$: then $T_j \rightarrow G$ at S_k. This implies that $T_j \rightarrow G$ at S_i, and thus $R_i \rightarrow G$ at S_i. As shown in 2.a, the server at S_k should ignore $w_i(d_0^k)$, and the inconsistency is resolved.

Suppose that $o_p = w_r(d_0^k)$. Because $R \rightarrow R_i$ at S_k, $R \rightarrow R_i$ at S_i. The resolution of the inconsistency between T_j and R_i is therefore reduced to resolving the inconsistency between T_j and R, but with fewer (i.e., p - 1) intermediate operations.

Suppose $o_p = r_l(d_0^k)$; the local execution resembles

E_k : .....o_j.....$o_{p-1} r_l(d_0^k) w_i^k(d_0^k)$.....

Suppose there exists $w_{t_0}(d_0^k) \in \mathcal{W}(T_0)$ between o_j and $r_l(d_0^k)$; then $T_0 \in \mathcal{GT} \cup \mathcal{RT}_i$. The problem is now reduced to the previous two cases with fewer intermediate operations.

Otherwise, the server is also able to ignore $w_i^k(d_0^k)$ without changing the semantics of the execution. The resulting execution is virtually equivalent to:

$\hat{E}_k$: .....$r_l(d_0^k) w_i^k(d_0^k) o_j$.....$o_{p-1}$.....

In any case, the inconsistency is either resolvable or reduced to another with fewer intermediate operations. Since p is finite, we conclude that any inconsistency between o_j and $w_i(d_0^k)$ is resolvable.

3. T_j follows R_i at site S_i, but precedes R_i at other sites, and $T_j \in RT_j$, where $j \neq i$. First, we observe that T_i and T_j do not directly conflict, because the replicated data sets of different sites are disjoint.

   As with case (2.b), all inconsistencies that result from indirect conflicts between T_i and T_j prove to be resolvable.