Deferred Update Protocols for Multi-Site Transactions

Parvathi Chundi *Daniel J. Rosenkrantz* *S. S. Ravi*

Department of Computer Science
University at Albany - State University of New York
Albany, NY 12222

February 21, 1996

Abstract

Several commercial distributed database systems provide an optional protocol that defers updates of replicas in order to attain higher transaction throughput. Each replicated data item is assigned a primary copy site, and has a set of sites with secondary copies. Typically, in a deferred update protocol, a transaction directly updates only the primary copy of each data item it modifies. After the transaction commits, the updates to primary copies by that transaction are sent transactionally to other sites containing secondary copies of these data items. Thus, the protocol allows the values of the primary and secondary copies of a data item to differ until the secondary copies are updated.

In an earlier paper, we investigated the deferred update approach for transactions that operate at only one site and then commit (with updates to secondary copies propagated after commitment). There, we showed that the serializability of transactions can be guaranteed if and only if the replication topology of the system is either a tree or a forest. In this paper, we discuss global transactions that operate at multiple sites in a system utilizing the deferred update approach. In such a system, a global transaction that updates replicated data items can use the underlying deferred update mechanism to ensure replica consistency. Such a transaction may commit after updating primary copies. The deferred update mechanism forwards the updates to sites containing secondary copies as usual. However, it is not clear if global serializability is ensured. We present two protocols to execute global transactions in a replicated database systems with a deferred update mechanism. The first protocol is applicable to systems whose replication topology is a tree. The second protocol is more general, and is applicable to systems whose replication topology is a forest. The protocols preserve the execution autonomy of the local sites and the underlying deferred update mechanism. They impose minimal overhead on transactions that operate at only one site. We show that both protocols ensure global serializability. We also propose a method for tolerating site failures.

1 Introduction

Replication of data is widely employed in distributed databases to enhance data availability, reliability and performance. Critical data is stored at several sites so that the system can continue to operate even when some of the sites fail. Also, improved query performance can be achieved by accessing the nearest copy. An important goal of any replicated database system is to make the fact that data is replicated transparent to the users. That is, transactions issue reads and writes on data items, and the system is responsible for translating these operations into read and write operations on one or more copies of these data items. Consequently, a replicated database system should behave like a single copy (non-replicated) database as far as the users are concerned. Ideally, the interleaved execution of transactions in a replicated database system should be equivalent to a serial execution of these transactions on a single copy database.

Transaction processing in replicated database systems has been studied by a number of researchers (see for example [AE90, AE92, BHG87, BG84, PL91] and the references contained therein). Several protocols achieving replica consistency, such as two phase commit and quorum consensus, have been proposed in the literature [BHG87, Gi79, Th79]. These protocols allow a transaction to access a replicated data item at a site only if the transaction also accesses a certain subset or a specified number of replicas of that data item. Some form of two phase commit is supported by almost all commercial distributed database system products available today [St95, Sc94] because it guarantees the consistency of replicated data and the serializability of transactions. Many commercial vendors [St95, Mo94, Co93, MP+93, Or93] characterize a system using two phase commit protocol as providing **tight consistency** or **synchronous distributed replication**. This is because the two phase commit protocol commits a transaction that updates a replicated data item only after *all* the sites containing a replica of that data item approve committing the transaction. Tight consistency has some drawbacks in practice [St95, Mo94, MP+93, Sc94]. In fact, many database vendors indicate that in numerous applications, two phase commit is impractical. For instance, quoting from [MP+93],

> "Synchronous distributed replication is attractive in theory, but fails in the real world. Its reliance on 100 percent system availability makes maintaining a productive level of transaction throughput for distributed replication impossible."

Reference [Go94] gives an overview of commercially available distributed database systems with replication capability. Several of these systems, including SYBASE System 10 [Sy, Mo94, Co93, MP+93], Oracle 7 [Or93], IBM Datapropagator Relational [Ib94], and CA-OpenIngres [Sc94], support a protocol for tight consistency and also include an *optional* protocol that *defers* the update of replicas. The deferred update protocols support **loose consistency** [Mo94,

MP+93] by allowing some of the copies to be inconsistent for some time. One of the main advantages of loose consistency is that it provides better responsiveness since the waiting operations associated with protocols such as two phase commit are avoided. A similar approach, called **asynchronous coherency control**, has been been studied in [PL91].

In the **deferred update** approach, a transaction can commit after updating only one copy of a replicated data item. After the transaction commits, the update is propagated asynchronously to the other copies. Reference [Go94] discusses several methods used by commercial database systems to implement deferred update. A classification of these approaches is provided in [CRR96]. The vendors of commercial database systems that provide deferred update [Ib94, Mo94, Co93, MP+93, Or93] discuss several applications for which a deferred update protocol seems preferable to two phase commit. However, they do not address the issue of serializability in those systems. In fact, [CRR96] provides examples of non-serializable scenarios in systems with a deferred update capability. The paper "Things Every Update Replication Customer Should Know" [Go95] also discusses the importance of serializability in a deferred update system and concludes that

> "Asynchronous update replication should only be used after carefully assessing the risks. Replication products which do not enforce serializability may not be appropriate for applications requiring transaction integrity."

In [CRR96], we studied the **primary copy approach** to deferred update; this approach is supported by several commercial database systems. In this approach, each replicated data item is assigned a site where the **primary copy** of the data item is stored. Copies of this data item appearing at other sites are referred to as **secondary copies**. Different data items may have primary copies at different sites and a given data item need not have a copy at every site. (The notions of primary and secondary copies have also been used in Distributed INGRES [St79, SN77].) A **replication server**[1] is included at each site; the replication servers cooperate to ensure the propagation of an update to all copies.

We formalized the primary copy approach using what we call the **single-site strict primary update (single-site spu)** protocol, and addressed the issue of serializability[2]. (The issue of serializability is also addressed in [Go95], but [Go95] does not provide a solution.) Under the single-site spu protocol, transactions operate at *only one site*. In a system using the single-site spu protocol, updates to a data item can be done only at its primary site. Secondary copies are read-only. For a transaction T to update a replicated data item, T must operate at the site S containing the primary copy of the data item. After T commits, the replication server at S

[1] The term "replication server" is a trademark of SYBASE, Inc.

[2] The terminology used in this paper differs slightly from that used in [CRR96]. This allows us to simplify the terminology used in the protocols developed here.

sends all the updates made by T on primary copies at S transactionally to all sites containing the corresponding secondary copies. Since the commit of a transaction updating a primary copy at a given site results in messages going to other sites, we call such an update message a **ripple message**. We refer to the subtransaction executed as a result of receiving a ripple message as a **ripple subtransaction**. Thus, the values of the primary and secondary copies of a data item may differ until the updates on the secondary copies are completed. The single-site spu protocol also allows a transaction executing at a site to read secondary copies located at that site.

In [CRR96], we showed that the distribution topology of the replicated data determines whether a system employing the single-site spu protocol ensures serializability. We modeled this topology by a directed graph that we call the **data placement graph (DPG)**. This graph represents the distribution of the primary and secondary copies of data items across sites. The graph has nodes for sites and edges for data flow from primary copies to secondary copies. Under standard assumptions concerning the nature of distributed database systems and the concurrency control mechanisms used at each site, we showed that global serializability in a system using the single-site spu protocol is ensured if and only if the DPG satisfies a certain acyclicity condition. For any configuration violating this acyclicity condition, the single-site spu protocol can produce non-serializable histories. These non-serializable histories are produced by the simplest type of transactions that can be envisioned for a distributed system with replication, namely transactions that operate at only one site and send out ripple messages. Thus, the acyclicity condition on DPGs is a very strong lower bound that a data configuration must meet in order to have any hope that serializability can be maintained while using the primary copy approach to deferred updates.

The protocol in [CRR96] supports transactions that operate at only one site, and send out ripples when committed. However, a distributed system should provide facilities that permit a transaction to operate at multiple sites. In this paper, we examine the issues involved in executing transactions operating at multiple sites in a system that incorporates the ripple mechanism of the single-site spu protocol. We call a transaction operating at more than one site a **multi-site global (global)** transaction. The transaction management scheme uses a **Global Protocol Manager (GPM)** to which global transactions are submitted. The GPM operates at one site. It submits the operations of a global transaction to various sites depending on the data items accessed, and eventually commits the transaction. Local sites execute the transactions submitted to them. These transactions may be subtransactions belonging to global transactions or transactions submitted locally that operate at only one site as before.

Using a deferred update capability may decrease the time needed to execute a global transaction which updates replicated data. In such a case, a global transaction G may commit after

updating only one copy (the primary copy, in the case of a primary copy approach) of each replicated data item. After G commits, the underlying deferred update mechanism sends the new values to other sites. However, the temporary inconsistency of replicas allowed by a deferred update ripple mechanism may give rise to non-serializable histories. In fact, in a system with a deferred update capability, it is easy to construct non-serializable histories if transactions can operate at multiple sites (see Section 2).

We present a transaction management scheme for executing global transactions in a system where each site incorporates the ripple mechanism of the single-site spu protocol. We address the issue of serializability for this scheme. We refer to this approach as a **multi-site strict primary update (multi-site spu) protocol**. Since the edges in the DPG represent the direction of "flow" of ripples between sites, if global transactions operate at different sites that are connected by a path in the DPG, the ripple subtransactions may introduce indirect conflicts between them. Hence, to ensure global serializability, the GPM may decide to execute global transactions at additional sites depending on the structure of the DPG. Reflecting the impact of the structure of the DPG, we present two multi-site spu protocols. The structure of the replication topology of any given system determines which of these two protocols should be used. We first present a **tree based multi-site spu protocol** for the case where the DPG corresponding to the system contains a single connected component. We then show how to extend the tree based multi-site spu protocol to the case where the DPG contains two or more connected components. We refer to the latter protocol as the **forest based multi-site spu protocol**. In both cases, we prove that global serializability is ensured. We present both protocols because the forest based protocol is more general, whereas the tree based protocol is simpler and more efficient when the DPG consists of a single connected component.

Both of our multi-site spu protocols allow local sites to execute the single-site spu protocol for transactions that operate at only one site. Hence, it is necessary that the data placement graph corresponding to the replication topology of the system satisfy the acyclicity condition from [CRR96] to ensure global serializability. By establishing the correctness of our protocols, we show that the acyclicity condition is sufficient to ensure global serializability even when the transaction mix includes multi-site global transactions.

We summarize the salient features of the multi-site spu protocols below.

1. Transactions are allowed to operate at more than one site.

2. The *execution autonomy* of local sites is preserved as far as possible. Each local concurrency control is unaware that global transactions are running at other sites, and is unaware of the GPM.

3. Only minimal changes are made to the ripple mechanism used at the local sites for ensuring replica consistency. The GPM is unaware of the ripple subtransactions that are exchanged among sites.

4. Transactions that operate at only one site are executed as efficiently as before. The examples in [Mo94, MP+93, Or93] motivating the deferred update approach suggest that in many applications of that approach, transactions typically operate at only one site (and then send out ripples). These transactions will not incur the extra overhead imposed on multi-site global transactions.

We note that preserving local autonomy and ensuring serializability renders the protocols as well as the correctness proofs nontrivial.

We also discuss how to handle arbitrary site failures and the failure of the GPM. A global transaction must commit or abort at all sites it operates. We ensure the atomicity of global transactions by using a technique similar to the redo approach [BGS92].

Preserving the execution autonomy of local sites while ensuring database consistency is a major goal of the multi-site spu protocol. Recently, several protocols (see [BS88, BGS92, DEK+93, GRS94, JDE+94] and the references contained therein) have been proposed to preserve autonomy of local databases in a multidatabase system while ensuring database consistency. In particular, [DEK+93] and [JDE+94] deal with multidatabases with replicated data. They propose protocols that both ensure consistency of data and preserve local autonomy, but under the assumption that the local databases do not share data items prior to the creation of the multidatabase. The multi-site spu protocols presented here execute global transactions in a system where the local databases have shared data and use the deferred update ripple mechanism to maintain replica consistency.

The remainder of this paper is organized as follows. Section 2 presents some potential non-serializable scenarios which illustrate some of the pitfalls that a multi-site spu protocol must avoid in order to achieve serializability. Section 3 presents the distributed data model and related definitions. Section 4 formalizes the tree based multi-site spu protocol. Section 5 presents the forest based multi-site spu protocol. Section 6 presents a method for ensuring consistency and atomicity in the presence of site failures. Section 7 concludes the paper.

2 Some Pitfalls to Avoid

Since deferred update allows loose consistency of replicas, it may also affect global transactions. For example, the execution of a global transaction executing at a set of sites that share data items may be influenced by replica inconsistency among sites. We now give several examples

of possible threats to serializability. Each example is accompanied by a data placement graph (DPG) D corresponding to the placement of primary and secondary copies in the system. Each node in D represents a site. D contains an edge $S_i \rightarrow S_j$ if there is at least one data item whose primary copy is at S_i and for which there is a secondary copy at S_j. Figures include, for each site, the set of data items that are involved in the non-serializable execution. Alongside each node, we give the implications for transaction serialization due to conflicts at that site. In the examples, we denote the primary copy of a data item d by d itself and denote a secondary copy of d by d'. The read (write) operation of a transaction T_i on a data item x is denoted by $r_i(x)$ ($w_i(x)$). The commit of T_i is denoted by c_i. Send and receive operations on a ripple produced by the transaction T_i are denoted by $send_i$ and $recvd_i$ respectively.

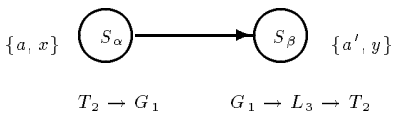

Figure 1: Indirect conflicts due to a single edge in the DPG

Example 2.1: (Figure 1) The data items at site S_α include $\{a, x\}$ and at site S_β include $\{a', y\}$. Here, a is replicated at both sites, while x is a local data item at S_α and y is a local data item at S_β. Transaction G_1 is a global transaction operating at sites S_α and S_β with the following operations $G_1 = r_1(x)\ w_1(x)\ w_1(y)\ c_1$. Transaction T_2 operates only at S_α, with operations $T_2 = r_2(x)\ w_2(a)\ c_2$. Transaction L_3 operates only at site S_β with operations $L_3 = r_3(a')\ w_3(y)\ c_3$.

Since G_1 is as global transaction, it is submitted to the GPM and its execution is coordinated by the GPM. However, T_2 and L_3 are not submitted to the GPM, so the GPM is unaware of them. Note that commit of transaction T_2 at site S_α results in a ripple message to be sent to site S_β. When this ripple message is received at S_β, a ripple subtransaction is submitted by the replication server at S_β to update the value of a'.

We have transactions G_1 and T_2 executing at site S_α. Let h_α be the history corresponding to the execution of these transactions at S_α, where $h_\alpha = r_2(x)\ w_2(a)\ c_2\ send_2\ r_1(x)\ w_1(x)\ c_1$. This introduces the serialization graph edge $T_2 \rightarrow G_1$ at S_α.

We have G_1, L_3 and the ripple subtransaction corresponding to the ripple message of T_2 from S_α executing at S_β. Let h_β be the history at S_β where $h_\beta = w_1(y)\ c_1\ r_3(a')\ w_3(y)\ c_3\ recvd_2\ w_2(a')\ c_2$. Hence, the local serialization graph at S_β contains the path $G_1 \rightarrow L_3 \rightarrow T_2$.

The global serialization graph contains a cycle $G_1 \rightarrow L_3 \rightarrow T_2 \rightarrow G_1$ involving G_1, L_3 and T_2. □

In Example 2.1, conflicts are introduced between subtransactions belonging to G_1 executing at different sites due to the ripples from S_α to S_β. The following example illustrates that indirect

conflicts may also be introduced when a global transaction executes at sites which do not have a data placement edge between them, but which are connected by a path in the data placement graph.

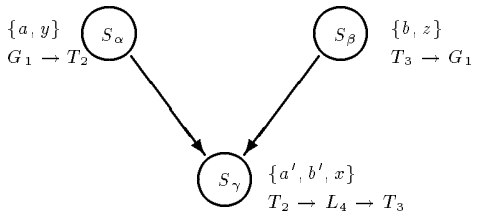

Figure 2: Indirect conflicts due to a path between sites

Example 2.2: (Figure 2) Site S_α includes data items $\{a, y\}$, site S_β includes data items $\{b, z\}$ and site S_γ includes data items $\{a', b', x\}$. Data item a is located at S_α and S_γ, with the primary copy at S_α. Data item b is located at S_β and S_γ with the primary copy at S_β. Data items x, y, and z are local data items at S_γ, S_α and S_β respectively.

G_1 is a global transaction submitted to the GPM, with operations $G_1 = r_1(a)\ w_1(y)\ r_1(b)\ w_1(z)\ c_1$. Transaction T_2 is submitted to S_α, with operations $T_2 = r_2(y)\ w_2(a)\ c_2$. Transaction T_3 is submitted to S_β, with operations $T_3 = r_3(z)\ w_3(b)\ c_3$. L_4 is submitted at site S_γ, with operations $L_4 = r_4(a')\ r_4(b')\ w_4(x)\ c_4$. Note that commit of T_2 (T_3) at S_α (S_β) generates a ripple message which is sent to S_γ (S_γ). The GPM executes G_1 at S_α and S_β.

At S_α, G_1 and T_2 execute. Let h_α be corresponding history where $h_\alpha = r_1(a)\ w_1(y)\ c_1\ r_2(y)\ w_2(a)\ c_2\ send_2$. The local serialization graph at S_α contains the edge $G_1 \rightarrow T_2$.

Let h_β be the history at S_β containing the operations belonging to G_1 and T_3, where $h_\beta = r_3(z)\ w_3(b)\ c_3\ send_3\ r_1(b)\ w_1(z)\ c_1$. This introduces the serialization graph edge $T_3 \rightarrow G_1$ at S_β.

Finally, let h_γ be the history at S_γ where $h_\gamma = recvd_2\ w_2(a')\ c_2\ r_4(a')\ r_4(b')\ w_4(x)\ c_4\ recvd_3\ w_3(b')\ c_3$. This introduces the following serialization graph edges at S_γ: $T_2 \rightarrow L_4 \rightarrow T_3$.

Hence, we have the following cycle in the serialization graph: $G_1 \rightarrow T_2 \rightarrow L_4 \rightarrow T_3 \rightarrow G_1$. □

It is clear from the above examples that the deferred update ripple mechanism creates interference among global transactions which may result in non-serializable executions. This implies that even if two global transactions do not execute together at any site, there may still be a path between them in the global serialization graph. One such scenario is presented below.

Example 2.3 (Figure 3) Site S_α includes data items $\{a, t\}$, site S_β includes data items $\{a', x\}$. Site S_γ includes data items $\{b, y\}$ and site S_δ includes data items $\{b', z\}$. Data items t, x, y, z are local data items at S_α, S_β, S_γ, and S_δ respectively. Data item a is replicated at S_α and S_β, with the primary copy at S_α. Data item b is replicated at S_γ and S_δ with the primary copy at S_γ.

G_1 and G_2 are global transactions, with operations $G_1 = r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)\ c_1$ and $G_2 = r_2(t)\ w_2(t)\ r_2(z)\ w_2(z)\ c_2$.

Note that G_1 must operate at sites S_β and S_γ, and G_2 must operate at sites S_α and S_δ. Transactions T_3 and T_4 are submitted at S_α and S_γ respectively. Transaction T_3 contains operations $T_3 = r_3(t)\ w_3(a)\ c_3$. Transaction T_4 contains operations $T_4 = r_4(y)\ w_4(b)\ c_4$. Commit of T_3 (T_4) at S_α (S_γ) causes a ripple message to be sent to S_β (S_δ). Finally, transactions L_5 and L_6 are submitted to S_β and S_δ respectively where $L_5 = r_5(x)\ r_5(a')\ c_5$ and $L_6 = r_6(z)\ r_6(b')\ c_6$.

At site S_α, the transactions G_2 and T_3 execute with history $h_\alpha = r_3(t)\ w_3(a)\ c_3\ send_3\ r_2(t)\ w_2(t)\ c_2$. The local serialization graph at S_α contains the edge $T_3 \rightarrow G_2$.

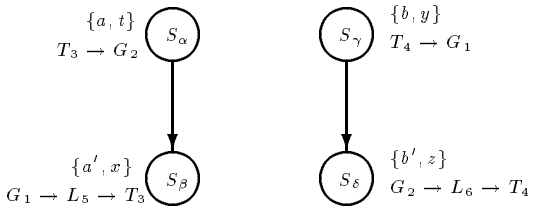

Figure 3: Indirect conflicts in a multi-component DPG

At S_β, G_1 and L_5 and the ripple subtransaction of T_3 execute. The history at S_β is $h_\beta = r_1(x)\ w_1(x)\ c_1\ r_5(x)\ r_5(a')\ c_5\ recvd_3\ w_3(a')\ c_3$. The local serialization graph at S_β contains the path $G_1 \rightarrow L_5 \rightarrow T_3$.

G_1 and T_4 execute at S_γ. The execution of T_4 results in a ripple message which is sent to S_δ. The history at S_γ is $h_\gamma = r_4(y)\ w_4(b)\ c_4\ send_4\ r_1(y)\ w_1(y)\ c_1$. The local serialization graph at S_γ contains the edge $T_4 \rightarrow G_1$.

At S_δ, we have the transactions G_2, L_6 and the ripple subtransaction corresponding to the ripple from T_4 at S_γ. The history at S_δ is $h_\delta = r_2(z)\ w_2(z)\ c_2\ r_6(z)\ r_6(b')\ c_6\ recvd_4\ w_4(b')\ c_4$. The local serialization graph at S_δ contains the path $G_2 \rightarrow L_6 \rightarrow T_4$.

The serialization graph edges at the local sites result in the following cycle in the global serialization graph: $T_3 \rightarrow G_2 \rightarrow L_6 \rightarrow T_4 \rightarrow G_1 \rightarrow L_5 \rightarrow T_3$. □

3 System Design

3.1 The Global System Architecture

A **replicated database system (RDBS)** consists of a set of **sites** $S_1, S_2, \ldots, S_n$ ($n > 2$). Each site includes a pre-existing **local database system (LDBS)**. Each pair of sites can communicate using a fifo discipline. Each site contains a **protocol manager** that communicates with other sites to maintain the consistency of replicated data. The RDBS contains a **Global Protocol**

Manager (GPM) which runs at one site and coordinates transactions accessing data at multiple sites. Each replicated data item has a designated site where the **primary copy** of the data item resides; the copies of this data item at other sites are **secondary copies**. Data items occurring at more than one site are referred to as **global data items**. Each site may also have **local data items**. We denote a secondary copy of a data item d by d' and the primary copy by d itself.

3.2 Data Placement Graph

The **data placement graph (DPG)** for a given RDBS is defined as follows. Each node in the DPG represents a site. There is a directed edge from S_i to S_j if there is at least one data item for which S_i is the primary site and S_j is a secondary site. Given a DPG, the corresponding **undirected data placement graph (UDPG)** is obtained by simply erasing the directions on the edges and combining multi-edges (if any) between a pair of vertices into a single edge. In a DPG, a pair of edges of the form (u, v) and (v, u) which form a directed cycle of length 2 will be referred to as **dual edges**. We say that a DPG $D(V, A)$ is **strongly acyclic** if it does not have dual edges and the undirected graph obtained from D by deleting the direction on each edge and combining the multi-edges into a single edge is acyclic.

3.3 The Single-site spu protocol

The local sites in the RDBS rely on the ripple mechanism of the single-site spu protocol to maintain replica consistency. Reference [CRR96] formalizes the single-site spu protocol and discusses the conditions under which the single-site spu protocol ensures serializability. We summarize the protocol and these conditions here.

The single-site spu protocol allows transactions to operate at only one site. A transaction is a partial order of read and write operations (denoted by r and w respectively) ending with a commit (c) or an abort (a) operation. As in Section 2, the read and write of a transaction T_i on a data item d are denoted by $r_i(d)$ and $w_i(d)$ respectively. Each LDBS executes two types of transactions.

1. A **local** transaction operates at a single site. It can read and write local data items at that site. It can read, but not write, global data items at that site.

2. A **single site primary (ssp)** transaction operates at a single site. It can read and write local and primary data items located at that site. Further, it can read, but not write, secondary data items at that site.

Whenever an ssp transaction at a site S_i updates primary data items and commits, the protocol manager at S_i sends the new values of these data items as a ripple message to the

corresponding secondary sites. When the protocol manager at a site S_j receives a ripple message, it executes a ripple subtransaction to update the corresponding secondary data items at S_j. The send and receive operations of a ripple generated by an ssp transaction T_k are denoted by $send_k(\Delta)$ and $recvd_k(\Delta)$ respectively, where Δ is the set of new values of primary copies modified by T_k. The ripple messages are sent in the order in which ssp transactions commit at S_i. Also, for each edge $S_i \rightarrow S_j$, the protocol manager at S_j submits the ripple subtransactions corresponding to ripple messages received from S_i in the order in which the ripple messages are received, so that each such ripple subtransaction is committed before the next one begins.

We define the local history at a site as a partial order over operations of all transactions executing at that site, including send and receive operations carried out by the protocol manager at that site. The partial order at each site must satisfy the usual conditions listed in [BHG87]. We assume that each LDBS uses strict two phase locking to produce serializable histories. Note that two phase locking provides a *pessimistic concurrency control*. Therefore, an LDBS does not abort a transaction that finished all its accesses, except in the event of a failure. We also assume that the ripple mechanism ensures that a ripple subtransaction is always committed. If a ripple subtransaction gets aborted, it is retried. A global history is a union of local histories at each site, augmented by the requirement that the send of a message occurs prior to the corresponding receive. We define a **spu-global history** to be a global history produced by the single-site spu protocol. In [CRR96], it is shown that the serializability of spu-global histories is based on the DPG corresponding to the RDBS. There we define a DPG to be a **spu-global serializable** if it has the property that every history produced by the single-site spu protocol executing in a data configuration corresponding to the DPG is globally serializable. Then, we prove that a DPG is spu-global serializable if and only if it is strongly acyclic.

3.4 Global serializability

Given a global history produced by executing a set of transactions on an RDBS, each site has a local serialization graph. The **global serialization graph** is obtained by taking the union of nodes and edges of the local serialization graph at each site in the RDBS. In the union, all subtransactions of a global transaction are combined into a single node and each ripple subtransaction is combined with the node that caused it. A global history is serializable if and only if the corresponding global serialization graph is acyclic.

4 The Tree Based Multi-Site Spu Protocol

In this section, we describe a multi-site spu protocol, which we call the **tree based multi-site protocol** for a system whose DPG consists of a single connected component. We require that

the DPG is strongly acyclic, and hence the UDPG is a tree.

The GPM uses the tree based multi-site spu protocol to execute global transactions which operate at two or more sites in the RDBS. A global transaction can read and write any (logical) data item in the RDBS. The GPM coordinates the execution of global transactions. In addition, the GPM also maintains a directory of local data items and primary and secondary copies of global data items. When a global transaction G_i is submitted to the GPM, the GPM first decides the set of sites at which G_i must be executed. Then, it creates a **server** for G_i at each of these sites. Each such server interacts with the LDBS to execute G_i's operations. A subtransaction of G_i at a site S_j is the partial order of operations executed at S_j that belong to G_i. Figure 4 illustrates the architecture of the system.

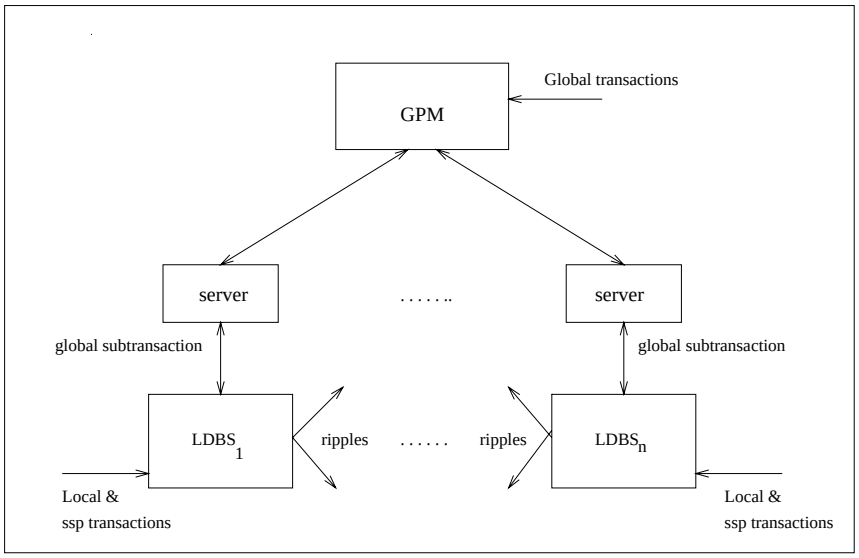

Figure 4: RDBS Architecture

4.1 Counter data items

When a primary site and a secondary site share data items, we can say that the secondary site stores a (possibly lagging) "view" of the database at the primary site. A global transaction operating at both sites might potentially see inconsistent views; we detect such inconsistencies using *view ids*. We create new global data items, which we refer to as **counter** data items. (The counter data items are related to the concept of *tickets* in [GRS94].) A counter data item serves as a *view id*. We create a unique view id at a primary site for each of it secondary sites. A secondary copy of the view id is stored at the corresponding secondary site. Therefore, for each edge $S_i \rightarrow S_j$ in the DPG, we create a unique counter data item ct_{ij} that is replicated at S_i and S_j, with the primary copy at S_i and the only secondary copy at S_j. If a transaction updates the primary copy of any data item at S_i that has a secondary copy at S_j, then the transaction is

augmented by two additional operations: read the current value of ct_{ij} and write a new value of ct_{ij} by incrementing it by 1. Since counter data items are also replicated data items, whenever the primary copy of a counter is modified, the corresponding ripple subtransaction updates the secondary copy.

4.2 The Protocol Description

In designing the multi-site spu protocol, we minimize the changes made to the local concurrency controls and the ripple mechanism. Consequently, we assume the following.

1. Ssp transactions are not coordinated by the GPM even though they update global (replicated) data.

2.The GPM is not aware of ripple messages that are generated by the protocol managers at various sites in the system.

3.Each subtransaction of a global transaction is treated as just another local or ssp transaction by each LDBS. In other words, the LDBS need not be aware of global transactions or the GPM.

A global transaction G_i is submitted to the GPM, accompanied by its read and write sets which we denote by $RS(G_i)$ and $WS(G_i)$. For the tree based multi-site spu protocol, the DPG corresponding to the RDBS contains a single connected component and is strongly acyclic. The execution of G_i proceeds in the following three phases.

Phase I: Construct the set of sites at which G_i must execute.

The GPM first creates the **original site set** of G_i, denoted by $oss(G_i)$. For each data item $x \in RS(G_i) \cup WS(G_i)$, if x is a local data item, then the site at which x is located is added to $oss(G_i)$. If x is a global data item and $x \in WS(G_i)$, then the primary site of x is added to $oss(G_i)$. If x is a global data item and $x \in RS(G_i)$ only, then any site containing a copy of x can be added to $oss(G_i)$. The GPM finds the minimal subtree Γ in the UDPG that includes all the sites in $oss(G_i)$. (Since the UDPG is a tree, this minimal subtree is unique.) We refer to the set of sites in Γ as the **extended site set** of G_i and denote it by $ess(G_i)$. The global transaction G_i is made to execute at all sites in $ess(G_i)$. Hence, the GPM creates a server at each of these sites for executing G_i.

Phase II: Augment G_i with operations on counters and submit G_i for execution.

For each site S_j in $ess(G_i)$, the subtransaction G_{ij} contains the union of the following sets of operations. We refer to these operations collectively as **data operations**.

• **original accesses**: If $S_j \in oss(G_i)$, then G_{ij} contains operations of G_i on some of the (non-counter) data items at S_j. Otherwise, the set of original accesses is empty.

• **counter accesses**: G_{ij} reads counter data items that S_j shares with its neighbor sites that belong to $ess(G_i)$. If G_i is updating primary data items at S_j, G_{ij} also contains the read and

write operations on the corresponding counter data items.

• **secondary accesses**: For each secondary data item d' at S_j, G_{ij} writes d' if $d \in WS(G_i)$.

Note that if G_i updates a primary data item, it updates only those secondary copies that are located at some site in $ess(G_i)$.

The GPM submits a data operation α of G_i to the corresponding server after all the data operations that α depends on have been completed. We denote this submission by $submit(\alpha)$. After submitting α, the GPM waits for a notification message carrying success/failure from the server. If the GPM receives failure or times out waiting for the notification message, it aborts G_i. (A timeout may also mean that G_i is involved in a deadlock.) We denote the send and receive of a notification message by $send\text{-}notification(\delta)$ and $recv\text{-}notification(\delta)$ respectively where δ is the set of values returned. Along with success/failure, the notification message also reports the values returned by the read operations of G_i. This allows data flow between data operations of G_i executing at different sites. Using these reported values, the GPM aborts G_i if it reads different values for a counter data item from two different sites.

We say that a global transaction enters the **finished** stage when the GPM finishes executing all of its data operations.

Phase III: Committing G_i.

For each site S_j in the RDBS, the GPM maintains a queue Q_j of global transactions. After G_i finishes executing data operations at all sites in $ess(G_i)$, the GPM adds G_i at the end of the queue Q_k corresponding to each site S_k in $ess(G_i)$. When G_i reaches the beginning of some queue Q_k, the GPM sends a *commit* message to the server for G_i at S_k. After G_{ik} commits at S_k, the server at S_k sends a *commit-done*(G_i) message to the GPM. After receiving this message, the GPM removes G_i from Q_k. Note that the GPM adds each global transaction to the appropriate queues in the order in which they enter the finished stage.

A global transaction G is said to be **committed** after the last *commit-done* message is received at the GPM. Note that G is said to be committed after it commits at all sites in $ess(G)$, so G does not wait for the ripple subtransactions generated on behalf of G to commit. A ripple subtransaction at a site S not in $ess(G)$ is submitted and committed at S whenever the ripple message is received by the protocol manager at S. On the other hand, if S is in $ess(G)$, a ripple message received as a result of the commit of G is discarded because the tree-structured multi-site spu protocol has already updated the secondary copies at S as part of G.

4.3 Selection of the Original Site Set

In Phase I of the protocol, if $RS(G_i)$ contains some global data items that are not in $WS(G_i)$, there may be some flexibility in choosing $oss(G_i)$. However, we can assume that G_i cannot

operate at only one site, since otherwise it would be more efficient to treat G_i as an ssp or a local transaction. As a consequence, it can be shown that there is a unique minimal site set that satisfies the requirements of $ess(G_i)$. More precisely, for any nonempty set of sites $\mathcal{S}$, let $ess(\mathcal{S})$ denote sites in the unique minimal subtree that includes all sites in $\mathcal{S}$. Let us say that $\mathcal{S}$ **satisfies the access requirements** of G_i if using $\mathcal{S}$ as $oss(G_i)$ permits all of the data operations of G_i to be performed. Then, there is a unique set of sites $\mathcal{T}$ such that for all sets $\mathcal{S}$ satisfying the access requirements of G_i, $\mathcal{T} \subseteq ess(\mathcal{S})$. Further, there is a set $\mathcal{S}$ satisfying the access requirements of G_i, such that $\mathcal{T} = ess(\mathcal{S})$.

4.4 Discussion

We now show how the first two hypothetical non-serializable executions given in Section 2 are handled by the multi-site spu protocol.

Example 2.1: Since S_α contains some primary data items whose secondary values are located at S_β there is a counter, say $ct_{\alpha\beta}$, whose primary copy is at S_α, and whose only secondary copy is at S_β. Global transaction G_1 accesses data from sites S_α and S_β and hence $ess(G_1)$ includes S_α and S_β. Then, according to the protocol, the subtransaction of G_1 at S_α reads the value of the counter $ct_{\alpha\beta}$ at S_α. Similarly, the subtransaction of G_1 executing at S_β reads $ct'_{\alpha\beta}$ from S_β. Consider how the scenario given in Section 2 would be expanded to include the accesses on counter $ct_{\alpha\beta}$. At S_α, G_1 reads the value of $ct_{\alpha\beta}$ written by the ssp transaction T_2. At S_β, G_1 reads the value of $ct'_{\alpha\beta}$ before the ripple subtransaction T_2 is executed. Since each ssp transaction increments the value of the counter, the value of $ct_{\alpha\beta}$ read by G_1 at S_α is different from that read by G_1 at S_β. Then, according to the protocol, G_1 is aborted. □

Example 2.2: Global transaction G_1 operates at S_α and S_β. Since there is a primary data item at S_α whose secondary copy is at S_γ, there is a counter $ct_{\alpha\gamma}$ whose primary copy is at S_α and whose only secondary copy is at S_γ. Similarly, there is a counter $ct_{\beta\gamma}$ whose primary copy is at S_β and whose only secondary copy is at S_γ. Since $oss(G_1) = \{S_\alpha, S_\beta\}$, the unique subtree connecting these sites in the UDPG includes S_γ. Therefore, according to the protocol, $ess(G_1) = \{S_\alpha, S_\beta, S_\gamma\}$. G_1 reads $ct_{\alpha\gamma}$ from S_α and S_γ and $ct_{\beta\gamma}$ from S_β and S_γ. In the scenario described in the example, at S_α, G_1 reads the value of $ct_{\alpha\gamma}$ before T_2 updates it. At S_β, G_1 reads the value of $ct_{\beta\gamma}$ after it is written by T_3.

Since the ripple subtransactions T_2 and T_3 update the value of $ct'_{\alpha\gamma}$ and $ct'_{\beta\gamma}$ respectively and G_1 reads these data items, G_1 (directly or indirectly) conflicts with both T_2 and T_3 at S_γ. For the protocol to commit G_1, G_1 must read the same values for $ct'_{\alpha\gamma}$ and $ct'_{\beta\gamma}$ at S_γ as at S_α and S_β respectively. These reads at S_γ add edges G_1 to T_2 and from T_3 to G_1 in the local serialization graph at S_γ. These edges, along with the path $T_2 \rightarrow L_4 \rightarrow T_3$ would cause a cycle

in the serialization graph at S_γ, and hence the local concurrency control at S_γ restarts one of these transactions. □

5 The Forest Based Multi-site Spu Protocol

5.1 The Protocol Description

In this section, we describe the *forest based multi-site spu protocol*. This protocol is designed for the case where the DPG contains more than one connected component. We require that the DPG is strongly acyclic, so the UDPG is a forest.

The forest based multi-site spu protocol is an extension of the tree based multi-site spu protocol described in Section 4. When a global transaction G is executing in more than one connected component in the DPG, the forest based protocol may add more sites to $ess(G)$ than the tree based protocol. Therefore, the execution of global transactions by the tree based protocol is more efficient since their extended site sets may be smaller. Phases II and III of the forest based multi-site spu protocol are the same as those of the tree based protocol. Phase I of the forest based protocol is given below. As before, it is assumed that a global transaction G_i is submitted along with its read and write sets, $RS(G_i)$ and $WS(G_i)$.

Phase I: Construct $oss(G_i)$ as before. Let us assume that the sites in $oss(G_i)$ belong to the connected components C_1, C_2, ..., C_p in the UDPG.

1. For each l $(1 \leq l \leq p)$, let Ω_l be the following set of global transactions. Set Ω_l contains the global transaction that is the last one to finish within C_l and Ω_l contains all global transactions that are executing at some site in C_l and have not yet committed.

2. For each l $(1 \leq l \leq p)$, let $\Omega'_l = \Omega_l \cup \{G_m \mid \exists j$ such that $G_m \in \Omega_j$ and G_m operated at some site in $C_l\}$.

3. For each l $(1 \leq l \leq p)$, let Γ_l be the minimal subtree in the UDPG that includes all sites in $oss(G_i)$ that belong to C_l and includes, for each G_m in Ω'_l, at least one site from $ess(G_m)$. (Since each connected component of the UDPG is a tree, this minimal subtree is unique.)

4. Let $ess(G_i) = \cup_{l=1}^{p} \Gamma_l$.

5.2 Discussion

We now explain how the forest based multi-site spu protocol takes care of the scenario described in Example 2.3. The DPG in this case contains two connected components, namely $C_1 = \{S_\alpha, S_\beta\}$ and $C_2 = \{S_\gamma, S_\delta\}$.

Let us assume that G_1 arrived earlier than G_2 in the system. Since G_1 is the first global transaction to execute, $ess(G_1) = \{S_\beta, S_\gamma\}$. When G_2 arrives, G_1 either has not yet committed or is the last committed transaction in C_1 and C_2.

In Phase I of the execution of G_2, the protocol does the following. Site set $oss(G_2)$ is $\{S_\alpha, S_\delta\}$. Therefore, G_2 operates in C_1 and C_2. Step 1 sets $\Omega_1 = \{G_1\}$ and $\Omega_2 = \{G_1\}$. Step 2 sets $\Omega_1' = \{G_1\}$ and $\Omega_2' = \{G_1\}$. Step 3 sets $\Gamma_1 = \{S_\alpha, S_\beta\}$ and $\Gamma_2 = \{S_\gamma, S_\delta\}$. Therefore, G_2 executes at all four sites in the system.

Now, consider C_1. There is a counter data item $ct_{\alpha\beta}$ whose primary copy is located at S_α with a secondary copy at S_β. Since G_2 operates at both these sites, it reads $ct_{\alpha\beta}$ from both sites. According to the example, G_2 reads the after value of T_3 at S_α. For G_2 to commit, it must read the same value of $ct'_{\alpha\beta}$ at S_β. This means that there is an edge from T_3 to G_2 in the local serialization graph at S_β; and consequently a path from G_1 to G_2. Therefore, G_1 must commit at S_2 and release its locks before G_2 performs the read on $ct'_{\alpha\beta}$ at S_β. Therefore, G_2 is waiting for G_1 at S_β.

In connected component C_2, S_γ and S_δ share a counter data item $ct_{\gamma\delta}$ which G_2 reads at both S_γ and S_δ. Since G_2 reads $ct'_{\gamma\delta}$ at S_δ before T_4 updates it, it must also read $ct_{\gamma\delta}$ at S_γ before T_4 updates it. Therefore, there is an edge from G_2 to T_4 in the local serialization graph at S_γ, and hence a path from G_2 to G_1 at S_γ. For G_1 to perform its operations at S_γ, G_2 must commit at S_γ and release its locks. Therefore, G_1 is waiting for G_2 at S_γ. Hence, we have a deadlock situation and the GPM aborts one of the global transactions (whichever times out earlier). □

6 Handling Failures

In this section, we discuss how to ensure database consistency and atomicity of transactions in the presence of failures. We explain how to handle arbitrary site failures and the failure of the GPM. When a local site S fails, all transactions executing at S that have not yet committed at the time of failure are aborted. Since local and ssp transactions operate at only one site, aborting these transactions does not violate atomicity. However, ensuring atomicity of ssp transactions requires that if an ssp transaction T commits at a site S, the ripple subtransactions sent to other sites as a result must also commit. Similarly, The GPM must make sure that a global transaction either commits or aborts at all sites. Furthermore, if a global transaction commits, its ripple subtransactions must also commit. (This is a part of the ripple mechanism.)

6.1 Atomicity of global transactions

We first discuss how to handle arbitrary site failures. Handling site failures while ensuring local autonomy has been studied in [BGS92, SKS91, KLS90, WV90]. Our focus is on ensuring atomicity of global transactions in a system using a multi-site spu protocol. A global transaction may be aborted at a site by the GPM, or by the local concurrency control when it is involved in a local deadlock. In such a case, the GPM must abort the global transaction at all other sites.

Suppose a global transaction G completes its data operations at all sites and the GPM decides to commit it. Since we assume that local sites use pessimistic concurrency controls (such as two phase locking) and G has completed its data operations, G will not be aborted by a local site for deadlock reasons. However, it may still be aborted in case of a local site failure. Suppose global transaction G executing at a set of sites is aborted at some site S due to local site failure, but is committed at all other sites. After S recovers, atomicity can be achieved using one of the following approaches [BGS92].

redo: The writes of the aborted subtransaction of G at S are installed by executing a *redo* transaction consisting of all the write operations of G at S.

retry: The entire aborted subtransaction including its reads and writes is executed again at S.

compensate: At each site where G committed, a compensating transaction is executed to semantically undo the effects of G.

We use the redo method to ensure global atomicity in the presence of failures. To ensure that G either commits at all sites or aborts at all sites, the GPM uses an atomic commit protocol such as two phase commit. This two phase commit is done the sites at which G_i operates. We modify Phase III of the protocol as follows. Let G_i be a global transaction with extended site set $ess(G_i)$. After G_i performs its data operations at all sites in $ess(G_i)$, the GPM does the following.

Phase III:

1. The GPM sends a *prepare-to-commit* message to the server created for G_i at each site in $ess(G_i)$. When a server receives the *prepare-to-commit* message from the GPM, it votes to either commit or abort G_i. If a server votes to commit the transaction, it enters a *prepared state* for G_i.

2. If the GPM receives commit votes from all the servers, it decides to commit G_i. It records this decision on its log. Then, for each site $S_j \in ess(G_i)$, the GPM adds G_i to the end of queue Q_j. When G_i reaches the head of queue Q_j, the GPM sends a *commit* message to the server.

3. When a server receives a *commit* message from the GPM, it submits a commit to the local site. After the commit is done, it sends a *commit-done* message to the GPM.

4. If the GPM receives an abort vote or times out waiting for a vote from some server, it decides to abort G_i and sends an *abort* message to all servers created for G_i.

For a two phase commit scheme to work, when a server enters the prepared state, it must be in a position to comply with the decision of the GPM, even in the presence of failures. Since it is possible for a site to fail while a server is in the prepared state, the server at a site must store the updates made by G_i onto a stable storage *before* entering the prepared state. If the GPM decides to commit G_i, but G_{ij} gets aborted because of a site failure at S_j, then after S_j recovers,

G_{ij} can be redone by installing the updates from the stable storage.

If the LDBSs provide a local prepare-to-commit operation that allows a server to force transaction log records into a stable storage and to abort or commit a transaction only after the GPM makes that decision, then servers can make use of that operation and easily achieve atomicity. So, we consider the case where the LDBSs do not provide such an operation. Reference [BST90] discusses a method for ensuring atomicity and database consistency under failures in cases where the local sites do not support prepare-to-commit operation locally. The method discussed in that paper places restrictions on the accesses by global and local transactions so that consistency of data can be guaranteed despite failures in the system. One way to avoid placing restrictions on the accesses by transactions is to simulate the prepared-to-commit operation. We propose here a method that provides a prepared-to-commit state for global transactions. At each site S in the RDBS, the system includes an **Auxiliary Recovery Manager (ARM)** that maintains a stable log, which we call the **auxiliary log**, for global transactions operating at S. The auxiliary log, as opposed to a traditional write-ahead log [BHG87], contains only the "redo" information. This is because, when a site fails before a global transaction commits, the global subtransaction executed at that site is rolled back, thereby obviating the need for undo information in the auxiliary log.

Step 1 of Phase III described above is modified as follows. Suppose the server for global transaction G_i at a site S_j receives a *prepare-to-commit* message from the GPM and decides to commit G_i at S_j. The server requests the ARM at S_j to store the updates by G_{ij} along with its transaction id in the auxiliary log. The server then enters the prepared state by voting to commit G_i. Step 2 is unchanged. In Step 3, after G_i is committed at S_j, the server for G_i at S_j requests the ARM to write a *commit-complete(G_i)* entry in the auxiliary log, and then sends a *commit-done* message to the GPM. In Step 4 of the protocol, if the GPM's decision is to abort G_i, then the server informs the ARM to delete the log entry for G_i.

Now, suppose S_j fails at some point in time. After S_j recovers, the local recovery manager restores the database. Then, the ARM examines the auxiliary log to see if any global transactions were in the prepared state when the crash occurred. For each G in the prepared state, the ARM sends a message to the GPM requesting the outcome of G. If the GPM decided to commit G, then the ARM obtains the new values for the data items updated by G from the auxiliary log, installs them and writes a *commit-complete(G)* in the log, thereby committing G at S_j. If the GPM aborted G, the ARM deletes the log entry for G. If the GPM has not reached a decision about G_i, then the ARM votes to commit G_i at S_j. Steps 2,3 and 4 of Phase III are carried as before.

Note that if G_i commits at S_j, but the site failed before the ARM could write *commit-*

complete(G_i) in the auxiliary log, G_i will be redone. However, this does not cause inconsistencies because the redo writes the same values that were written before the crash.

6.2 Handling Failure of the GPM

Since the GPM acts as a coordinator for all global transactions, in event of its failure, global transactions may be aborted at local sites and no more global transactions can be executed until the GPM recovers. In order to deal with failure, the GPM maintains a stable log, which we call the **GPM log**. For each global transaction G executed, the GPM records in the log, $ess(G)$ and either a *commit* or an *abort* entry recording a decision to commit G (in Step 2 of the two phase commit protocol) or abort G (in Step 4). After a global transaction commits at all sites, the GPM writes a *transaction-complete* entry to the GPM log.

Each server sends an *are-you-alive* message to the GPM if it does not receive a message from the GPM for a certain period of time. If the GPM does not respond to the *are-you-alive* message, then the server senses failure of the GPM. In such a case, if it is not in prepared state, it aborts the global transaction it is executing at the local site and aborts itself.

If a server is in a prepared state when it senses the failure of the GPM, it cannot abort itself or the transaction G it is executing at the local site, since G may have committed at other sites. In this case, the servers may wait for the GPM to recover or use a protocol similar to the initiator's algorithm [BHG87] to recover from the failure. After the GPM recovers from the failure, it creates for each site S_j, an empty queue Q_j. It then inspects the GPM log. For each *commit* entry of a global transaction G_i that does not contain a corresponding *transaction-complete* entry in the GPM log, it adds G_i to each Q_j where $S_j \in ess(G_i)$. Note that the GPM adds global transactions to queues in the order in which their *commit* entries appear in the GPM log. The GPM commits these global transactions as before.

6.3 Atomicity of ripple subtransactions

There are many approaches to ensuring the atomicity of ripple subtransactions. We discuss one approach here. There are two major recovery issues involving ripple subtransactions. First, if a site running a ripple subtransaction fails, upon recovery, these ripple subtransactions must be performed. Second, if a site commits a transaction that updated a primary copy, and the site fails before sending out the required ripple messages, upon recovery, these ripple messages must be sent. Upon recovery, ripple messages are dealt with after the ARM completes the recovery of global transactions.

When a site S_α fails, the protocol manager at S_α may or may not fail. We deal with these two cases separately.

We first consider the case where the protocol manager does not fail when S_α fails. In this case, the protocol manager submits the ripple subtransactions for each ripple message that was received but not committed, and sends ripple messages (if any) to other sites.

The case where the protocol manager also fails when S_α fails is handled as follows. After S_α recovers, the protocol manager obtains the ripple messages sent to S_α from other sites, and then submits these ripple subtransactions. Let S_1, ..., S_l be the sites such that each secondary data item at S_α has its primary copy at some S_j $(1 \leq j \leq l)$. Then, there are secondary copies of counter data items $ct'_{1\alpha}$, ..., $ct'_{l\alpha}$ at S_α whose primary copies are located S_1, ..., S_l respectively. The protocol manager sends a *ripple-request* message to each of the S_j's. The *ripple-request* message includes the current value of $ct'_{j\alpha}$ at S_α. Next, the protocol manager at S_j forwards, in the proper order, all ripple messages to S_α that wrote a higher value for $ct_{j\alpha}$ than the value included in the *ripple-request* message. Finally, the protocol manager at S_α forms and submits these ripple subtransactions.

If a protocol manager at a site S_α fails and recovers, it may have no knowledge of the ripple messages it sent earlier. Let S_1, ..., S_l be the sites that contain secondary copies of data items whose primary copies are located at S_α. Then, during recovery, the protocol manager at S_α obtains the value of counter $ct'_{\alpha j}$ from each of the S_j $(1 \leq j \leq l)$ and sends each S_j ripple messages corresponding to the transactions that wrote a higher value of $ct_{\alpha j}$ at S_α and committed prior to the failure.

7 Conclusions

A deferred update approach is supported by several commercial database systems, such as SYBASE System 10, Oracle 7, CA-OpenIngres, and IBM Datapropagator Relational etc., to maintain replica consistency in an efficient manner. In a primary copy deferred update approach, each replicated data item is assigned a primary copy site, and the other copies are referred to as secondary copies. In an earlier paper [CRR96], we showed that database consistency is ensured in a system using the primary copy deferred update approach only if the corresponding data placement graph satisfies an acyclicity condition. In this paper, we have investigated the problem of executing global transactions that operate at multiple sites in a system utilizing the primary copy deferred update mechanism. We presented two protocols, one for tree-structured data placement graphs and the other for forest-structured data placement graphs. We showed that both of these protocols guarantee global serializability. Further, these protocols preserve execution autonomy of the concurrency control mechanisms at each site and introduce only a small overhead on transactions that operate at a single site. We also presented a method for tolerating arbitrary site failures and the failure of the GPM.

References

[AE90] D. Agrawal and A. El Abbadi, "Exploiting Logical Structures of Replicated Databases," *Inf. Proc. Lett.*, Vol. 33, No. 5, Jan. 1990, pp 255-260.

[AE92] D. Agrawal and A. El Abbadi, "The Generalized Tree Quorum Protocol: An Efficient Approach to Managing Replicated Data," *ACM TODS*, Vol. 17, No. 4, Dec. 1992, pp 689-717.

[BG84] P. A. Bernstein and N. Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM TODS*, Vol. 9, No. 4, Dec. 1984, pp 596-615.

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

[BS88] Y. Breitbart and A. Silberschatz, "Multidatabase Update Issues," *Proc. of 1988 SIGMOD Conf.*, Chicago, IL, June 1988, pp. 135-142.

[BST90] Y. Breitbart, A. Silberschatz, and G. Thompson, "Reliable Transaction Management in a Multidatabase System," *Proc. of the 1990 ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, pp. 215-224.

[BGS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz, "Overview of Multidatabase Transaction Management," *VLDB Journal*, Vol 2, 1992, pp 181-239.

[CRR96] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi, "Deferred Updates and Data Placement in Distributed Databases", *To appear in ICDE '96*, New Orleans, LA, Feb 1996.

[Co93] M. Colton, "Replicated Data in a Distributed Environment," *Proc. of 1993 ACM SIGMOD Conf.*, Washington, DC, May 1993, pp 464-466.

[DEK+93] W. Du, A. Elmagarmid, W. Kim, and O. Bukhres, "Supporting Consistent Updates in Replicated Multidatabase Systems", *VLDB Journal*, Vol. 2, No. 2, 1993.

[Gi79] D. K. Gifford, "Weighted Voting for Replicated Data," *Proc. 7th SOSP*, Dec. 1979, pp 150-159.

[Go95] R. Goldring, "Things every update replication customer should know", *Proc. of 1995 ACM SIGMOD Conf.*, San Jose, CA, May 1995, pp 439-440.

[Go94] R. Goldring, *A discussion of Relational Database Replication Technology*, InfoDB, Vol.8, No.1, Spring 1994.

[GRS94] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth, "Using Tickets to Enforce the Serializability of Multidatabase Transactions", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 1, Feb 1994, pp 166-180.

[Ib94] IBM, *An introduction to Datapropagator Relational*, Release 2, Technical Document, IBM, Dec 1994.

[JDE+94] Jin Jing, W. Du, A. Elmagarmid, and O. Bukhres, "Maintaining Consistency of Replicated Data in Multidatabase Systems", *Proc. International Conf. on Distributed Computing Systems*, 1994.

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz, " A Formal Approach to Recovery by Compensating Transactions", *Proc. of International Conf. on Very Large Databases*, Brisbane, Australia, 1990.

[Mo94] A. Moissis, "SYBASE Replication Server: A Practical Architecture for Distributing and Sharing Corporate Information," Technical document, SYBASE Inc., March 1994.

[MP+93] N. Monserrat, T. Palanca, M. Deppe and B. Hartman, "Replication Server: A Component of SYBASE System 10," Technical document, SYBASE Inc., April 1993.

[Or93] Oracle Corporation, "Oracle 7^{TM} Symmetric Replication: Asynchronous Distributed Technology," White paper, Sept. 1993.

[PL91] C. Pu and A. Leff, "Replica Control in Distributed Systems: An Asynchronous Approach," *Proc. 1991 ACM SIGMOD Conf.*, Denver, CO, May 1991, pp 377-386.

[St95] D. Stacey, "Replication: DB2, Oracle, or Sybase?", *SIGMOD Record*, Vol. 24, No. 4, Dec. 1995.

[Sc94] G. Schussel, "Database Replication: Playing Both Ends Against the Middleware," *Client/Server Today*, Nov.1994, pp 57-67.

[SKS91] N. R. Soparkar, H. F. Korth, and A. Silberschatz, "Failure-resilient Transaction Management in Multidatabases", *IEEE Computer*, 24(12), 1991, pp 28-36.

[St79] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. Soft. Engineering.*, Vol. SE-5, No. 3, May. 1979, pp 188-194.

[SN77] M. Stonebraker and E. Neuhold, "A Distributed Database Version of INGRES," *Proc. 2nd Berkeley Workshop on Distributed Databases and Computer Networks*, Berkeley, CA, May 1977, pp 19-36.

[Sy] Sybase Corporation, "SYBASE Replication Server Technical Overview", White Paper.

[Th79] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", *ACM TODS*, Vol. 4, No. 2, June 1979, pp 180-209.

[WV90] A. Wolski and J. Veijalainen, "2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase", *Proc. of the International Conference on Database Parallel Architectures and Their Applications*, Miami, FL, 1990