Chapter 13

Data Replication

As we discussed in previous chapters, distributed databases are typically replicated. The purposes of replication are multiple:

1. **System availability.** As discussed in Chapter 1, distributed DBMSs may remove single points of failure by replicating data, so that data items are accessible from multiple sources. Consequently, even when one or multiple sites are down, data may be accessible from other sites.
2. **Performance.** As we have seen in previous chapters, one of the major contributors to response time is the communication overhead. Replication enables us to locate the data closer to their access points, thereby localizing most of the access that contributes to a reduction in response time.
3. **Scalability.** As systems grow geographically and in terms of the number of data access points (consequently, in terms of the number of access requests), replication allows for a way to support this growth with acceptable response times.
4. **Application requirements.** Finally, replication may be dictated by the applications, which may wish to maintain multiple data copies as part of their operational specifications.

Although data replication has clear benefits, it poses the considerable challenge of keeping different copies synchronized. We will discuss this shortly, but let us first consider the execution model in replicated databases. Each replicated data item x has a number of copies $x_1, x_2, \ldots, x_n$. We will refer to x as the *logical data item* and to its copies (or *replicas*)[1] as *physical data items*. If replication transparency is to be provided, user transactions will issue read and write operations on the logical data item x. The replica control protocol is responsible for mapping these operations to reads and writes on the physical data items $x_1, \ldots, x_n$. Thus, the system behaves as if there is a single copy of each data item – referred to as *single system image* or *one-copy equivalence*. The specific implementation

[1] In this chapter, we use the terms “replica”, “copy”, and “physical data item” interchangeably.

of the Read and Write interfaces of the transaction monitor differ according to the specific replication protocol, and we will discuss these differences in the appropriate sections.

As discussed in Chapter 3, a distributed database may be fully or partially replicated. In the case of a partially replicated database, the number of physical data items for each logical data item may vary, and some data items may even be non-replicated. In this case, transactions that access only non-replicated data items are *local transactions* (since they can be executed locally at one site) and their execution typically does not concern us here. Transactions that access replicated data items have to be executed at multiple sites and they are *global transactions*.

When global transactions update copies of a data item at different sites, the values of these copies may be different at a given point in time. A replicated database is said to be in a *mutually consistent* state if all the replicas of each of its data items have identical values. What differentiates different mutual consistency criteria is how tightly synchronized replicas have to be; some ensure that replicas are mutually consistent when an update transaction commits (thus, they are usually called *strong consistency* criteria), while others take a more relaxed approach (and are referred to as *weak consistency* criteria).

A fundamental design decision in designing a replication protocol is the choice of *update management*: where the database updates are first performed, and how these updates are propagated to the other copies [Gray et al., 1996]. In terms of where updates are first performed, the techniques can be characterized as *centralized* if they perform updates first on a *master* copy, versus *distributed* if they allow updates over any replica[2]. In terms of how updates are propagated, the alternatives are identified as *eager* versus *lazy* techniques. Other design decisions involve data location, and number of copies; these are issues that we discussed in Chapter 3.

The combinations of various update management alternatives give rise to a family of *replication protocols*. The update propagation technique that is used in a protocol typically determines the mutual consistency criterion: eager protocols enforce strong mutual consistency while lazy ones enforce weak. All of the protocols would prefer to enforce serializable histories (what is referred to as *one-copy serializable* as discussed in Section 13.1.2), although weaker forms of transactional consistency have been proposed for performance reasons.

We discuss consistency issues in replicated databases in Section 13.1.1, and analyze the two dimensions of update management in Section 13.2. This will lead us to a discussion of the specific protocols in Section 13.3. In Section 13.4, we discuss the use of group communication primitives in reducing the messaging overhead of replication protocols. In these sections, we will assume that no failures occur so that we can focus on the replication protocols. We will then introduce failures and investigate how protocols are revised to handle failures (Section 13.5). Finally, in Section 13.6, we discuss how replication services can be provided in multidatabase systems (i.e., outside the component DBMSs).

[2]Centralized techniques are referred to, in the literature, as *single master*, while distributed ones are referred to as *multi-master* or *update anywhere*. These terms, in particular "single master" is confusing, since it refers to one alternative architecture for implementing centralized protocols. Thus, we prefer the more descriptive terms "centralized" and "distributed".

13.1 CONSISTENCY OF REPLICATED DATABASES

There are two issues related to consistency of a replicated database. One is mutual consistency, as discussed above, that deals with the convergence of the values of physical data items corresponding to one logical data item. The second is transaction consistency as we discussed in Chapter 11. Serializability, which we introduced as the transaction consistency criterion earlier, needs to be recast in the case of replicated databases. In addition, there are relationships between mutual consistency and transaction consistency, although they are separate. In this section we first discuss mutual consistency approaches and then focus on the redefinition of transaction consistency and its relationship to mutual consistency.

13.1.1 Mutual Consistency

As indicated earlier, mutual consistency criteria for replicated databases can either be strong or weak. Each class is suitable for different classes of applications with different consistency requirements.

Strong mutual consistency criteria require that all copies of a data item have the same value at the end of the execution of an update transaction. This is achieved by a variety of means, but the execution of 2PC at the commit point of an update transaction is a common way to achieve strong mutual consistency.

Weak mutual consistency criteria do not require the values of replicas of a data item to be identical when an update transaction terminates. What is required is that, if the update activity ceases for some time, the values *eventually* become identical. This is commonly referred to as *eventual consistency*, which refers to the fact that replica values may diverge, but will eventually converge. It is hard to define this concept formally or precisely, although the following definition is probably as precise as one can hope to get [Saito and Shapiro, 2005]:

> "A replicated [data item] is *eventually consistent* when it meets the following conditions, assuming that all replicas start from the same initial state.
>
> - At any moment, for each replica, there is a prefix of the [history] that is equivalent to a prefix of the [history] of every other replica. We call this a *committed prefix* for the replica.
> - The committed prefix of each replica grows monotonically over time.
> - All non-aborted operations in the committed prefix satisfy their preconditions.
> - For every submitted operation α, either α or [its abort] will eventually be included in the committed prefix."

It should be noted that this definition of eventual consistency – in particular the requirements that history prefixes are the same at any given moment and that the committed prefix grows monotonically – is rather strong. Many systems that claim to provide eventual consistency would violate these requirements. We leave this as an exercise for the readers.

Epsilon serializability (ESR) [Pu and Leff, 1991; Ramamritham and Pu, 1995] allows a query to see inconsistent data while replicas are being updated, but requires that the replicas converge to a one-copy serializable state once the updates are propagated to all of the copies. It bounds the error on the read values by an epsilon (ϵ) value (hence the name). ϵ is defined in terms of the number of updates (write operations) that a query "misses". Given a read-only transaction (query) T_Q, let T_U be all the update transactions that are executing concurrently with T_Q. If $RS(T_Q) \bigcap WS(T_U) \neq \emptyset$ (T_Q is reading some copy of some data items while T_U is updating (possibly a different) copy of those data items) then there is a read-write conflict and T_Q may be reading inconsistent data. The inconsistency (ϵ) is bounded by the changes performed by T_U. Clearly, ESR does not sacrifice database consistency, but only allows read-only transactions (queries) to read inconsistent data. For this reason, it has been claimed that ESR does not weaken database consistency, but "stretches" it [Wu et al., 1997].

While ϵ-serializability bounds the divergence in terms of the updates that copies miss, other looser bounds have also been discussed. It has even been suggested that users should be allowed to specify *freshness constraints* that are suitable for particular applications and the replication protocols should enforce these [Pacitti and Simon, 2000; Röhm et al., 2002a; Bernstein et al., 2006]. The types of freshness constraints that can be specified are the following:

- Time-bound constraints. Users may accept divergence of physical copy values up to a certain time: x_i may reflect the value of an update at time t while x_j may reflect the value at $t - \Delta$ and this may be acceptable.
- Value-bound constraints. It may be acceptable to have values of all physical data items within a certain a certain range of each other. The user may consider the database to be mutually consistent if the values do not diverge more than a certain amount (or percentage).
- Drift constraints on multiple data items. For transactions that read multiple data items, the users may be satisfied if the time drift between the update timestamps of two data items is less than a threshold (i.e., they were updated within that threshold) or, in the case of aggregate computation, if the aggregate computed over a data item is within a certain range of the most recent value (i.e., even if the individual physical copy values may be more out of sync than this range, as long as a particular aggregate computation is within range, it may be acceptable).

An important criterion in analyzing protocols that employ criteria that allow replicas to diverge is *degree of freshness*. The degree of freshness of a given replica r_i at time t is defined as the proportion of updates that have been applied at r_i (at time t) to the total number of updates [Pacitti et al., 1998, 1999].

13.1.2 Mutual Consistency versus Transaction Consistency

Mutual consistency, as we have defined it here, and transactional consistency as we discussed in Chapter 11 are related, but different. Mutual consistency refers to the replicas converging to the same value, while transaction consistency requires that the global execution history be serializable. It is possible for a replicated DBMS to ensure that data items are mutually consistent when a transaction commits, but the execution history may not be globally serializable. This is demonstrated in the following example.

Example 13.1

Consider three sites (A, B, and C) and three data items (x, y, z) that are distributed as follows: Site A hosts x, Site B hosts x, y, Site C hosts x, y, z. We will use site identifiers as subscripts on the data items to refer to a particular replica.

Now consider the following three transactions:

T_1:	$x \leftarrow 20$	T_2:	Read(x)	T_3:	Read(x)
	Write(x)		$y \leftarrow x + y$		Read(y)
	Commit		Write(y)		$z \leftarrow (x * y)/100$
			Commit		Write(z)
					Commit

Note that T_1's $Write$ has to be executed at all three sites (since x that is replicated at all three sites), T_2's $Write$ has to be executed at B and C, and T_3's $Write$ has to be executed only at C. We are assuming a transaction execution model where transactions can read their local replicas, but have to update all of the replicas.

Assume that the following three local histories are generated at the sites:

$$\begin{aligned}
H_A &= \{W_1(x_A), C_1\} \\
H_B &= \{W_1(x_B), C_1, R_2(x_B), W_2(y_B), C_2\} \\
H_C &= \{W_2(y_C), C_2, R_3(x_C), R_3(y_C), W_3(z_C), C_3, W_1(x_C), C_1\}
\end{aligned}$$

The serialization order in H_B is $T_1 \rightarrow T_2$ while in H_C it is $T_2 \rightarrow T_3 \rightarrow T_1$. Therefore, the global history is not serializable. However, the database is mutually consistent. Assume, for example, that initially $x_A = x_B = x_C = 10, y_B = y_C = 15$, and $z_C = 7$. With the above histories, the final values will be $x_A = x_B = x_C = 20, y_B = y_C = 35, z_C = 3.5$. All the physical copies (replicas) have indeed converged to the same value.

Of course, it is possible for both the database to be mutually inconsistent, and the execution history to be globally non-serializable, as demonstrated in the following example.

Example 13.2

Consider two sites (A and B), and one data item (x) that is replicated at both sites (x_A and x_B). Further consider the following two transactions:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x + 5$		$x \leftarrow x * 10$
	Write(x)		Write(x)
	Commit		Commit

Assume that the following two local histories are generated at the two sites (again using the execution model of the previous example):

$$\begin{aligned} H_A &= \{R_1(x_A), W_1(x_A), C_1, W_2(x_B), C_2\} \\ H_B &= \{R_2(x_B), W_2(x_B), C_2, W_1(x_A), C_1\} \end{aligned}$$

Although both of these histories are serial, they serialize T_1 and T_2 in reverse order; thus the global history is not serializable. Furthermore, the mutual consistency is violated as well. Assume that the value of x prior to the execution of these transactions was 1. At the end of the execution of these schedules, the value of x is 10 at site A while it is 6 at site B. Thus, in this example, the global history is non-serializable, **and** the databases are mutually inconsistent.

Given the above observation, the transaction consistency criterion given in Chapter 11 is extended in replicated databases to define *one-copy serializability*. One-copy serializability (1SR) states that the effects of transactions on replicated data items should be the same as if they had been performed one at-a-time on a single set of data items. In other words, the histories are equivalent to some serial execution over non-replicated data items.

Snapshot isolation that we introduced in Chapter 11 has been extended for replicated databases [Lin et al., 2005a] and used as an alternative transactional consistency criterion within the context of replicated databases [Plattner and Alonso, 2004; Daudjee and Salem, 2006]. Similarly, a weaker form of serializability, called *relaxed concurrency (RC-) serializability* has been defined that corresponds to "read committed" isolation level (Section 10.2.3) [Bernstein et al., 2006].

13.2 UPDATE MANAGEMENT STRATEGIES

As discussed above, the replication protocols can be classified according to when the updates are propagated to copies (eager versus lazy) and where updates are allowed to occur (centralized versus distributed). In this section we discuss these alternatives before we present protocols in the next section.

13.2.1 Eager Update Propagation

The eager update propagation approaches apply the changes to all the replicas within the context of the update transaction. Consequently, when the update transaction commits, all the copies have the same state. Typically, eager propagation techniques use 2PC at commit point, but, as we will see later, alternatives are possible to achieve agreement. Furthermore, eager propagation may use *synchronous* propagation of each update by applying it on all the replicas at the same time (when the $Write$ is issued), or *deferred* propagation whereby the updates are applied to one replica when they are issued, but their application on the other replicas is batched and deferred to the end of the transaction. Deferred propagation can be implemented by including the updates in the "Prepare-to-Commit" message at the start of 2PC execution.

Eager techniques typically enforce strong mutual consistency criteria. Since all the replicas are mutually consistent at the end of an update transaction, a subsequent read can read from any copy (i.e., one can map a $Read(x)$ to $Read(x_i)$ for any x_i). However, a $Write(x)$ has to be applied to all x_i (i.e., $Write(x_i), \forall x_i$). Thus, protocols that follow eager update propagation are known as *read-one/write-all* (ROWA) protocols.

The advantages of eager update propagation are threefold. First, they typically ensure that mutual consistency is enforced using 1SR; therefore, there are no transactional inconsistencies among replicas. Second, a transaction can read a local copy of the data item (if a local copy is available) and be certain that an up-to-date value is read. Thus, there is no need to do a remote read. Finally, the changes to replicas are done atomically; thus recovery from failures can be governed by the protocols we have already studied in the previous chapter.

The main disadvantage of eager update propagation is that a transaction has to update all the copies before it can terminate. This has two consequences. First, the response time performance of the update transaction suffers, since it typically has to participate in a 2PC execution and because the update speed is restricted by the slowest machine. Second, if one of the copies is unavailable, then the transaction cannot terminate since not all the copies need to be updated. As discussed in Chapter 12, if it is possible to differentiate between site failures and network failures, then one can terminate the transaction as long as only one replica is unavailable (recall that more than one site unavailability causes 2PC to be blocking), but it is generally not possible to differentiate these two types of failures.

13.2.2 Lazy Update Propagation

In lazy update propagation, the replica updates are not all performed within the context of the update transaction. In other words, the transaction does not wait until its updates are applied to all the copies before it commits – it commits as soon as one replica is updated. The propagation to other copies is done *asynchronously* from the original transaction, by means of *refresh transactions* that are sent to the replica sites some time after the update transaction commits. A refresh transaction carries the sequence of updates of the corresponding update transaction.

Lazy propagation is used in those applications for which strong mutual consistency may be unnecessary and too restrictive. These applications may be able to tolerate some inconsistency among the replicas in return for better performance. Examples of such applications are Domain Name Service (DNS), databases over geographically widely distributed sites, mobile databases, and personal digital assistant databases [Saito and Shapiro, 2005]. In these cases, usually weak mutual consistency is enforced.

The primary advantage of lazy update propagation techniques is that they generally have lower response times for update transactions, since the update transaction can commit as soon as it has updated one copy. The disadvantages are that the replicas are not mutually consistent and some replicas may be out-of-date, and, consequently, a local read may read stale data and does not guarantee to return the up-to-date value. Furthermore, under some scenarios that we will discuss later, transactions may not see their own writes, i.e., $Read_i(x)$ of an update transaction T_i may not see the effects of $Write_i(x)$ that was executed previously. This has been referred to as *transaction inversion* [Daudjee and Salem, 2004]. Transaction inversions can be prevented by providing “strong” transactional guarantees [Daudjee and Salem, 2004, 2006], but these may be expensive to provide.

13.2.3 Centralized Techniques

Centralized update propagation techniques require that updates are first applied at a master copy and then propagated to other copies (which are called *slaves*). The site that hosts the master copy is similarly called the *master site*, while the sites that host the slave copies for that data item are called *slave sites*.

In some techniques, there is a single master for all replicated data. We refer to these as *single master* centralized techniques. In other protocols, the master copy for each data item may be different (i.e., for data item x, the master copy may be x_i stored at site S_i, while for data item y, it may be y_j stored at site S_j). These are typically known as *primary copy* centralized techniques.

The advantages of centralized techniques are two-fold. First, application of the updates is easy since they happen at only the master site, and they do not require synchronization among multiple replica sites. Second, there is the assurance that at least one site – the site that holds the master copy – has up-to-date values for a data item. These protocols are generally suitable in data warehouses and other applications where data processing is centralized at one or a few master sites.

The primary disadvantage is that, as in any centralized algorithm, if there is one central site that hosts all of the masters – this site can be overloaded and become a bottleneck. Distributing the master site responsibility for each data item as discussed above is one way of reducing this overhead, but raises consistency issues, in particular with respect to maintaining global serializability since the refresh transactions have to be executed at the replicas in the same serialization order. We discuss these further in relevant sections.

13.2.4 Distributed Techniques

In distributed techniques, the update is applied on the local copy at the site where the update transaction originates and then the updates are propagated to the other replica sites. These are called distributed techniques since different transactions can update different copies of the same data item located at different sites (they are also referred to as multi-master techniques, since any copy where an update transaction originates can perform the role of a master). They are appropriate for collaborative applications with distributive decision/operation centers. They can more evenly distribute the load, and may provide the highest system availability if coupled with lazy propagation techniques.

A serious complication that arises in these systems is that different replicas of a data item may be updated at different sites (masters) concurrently. If distributed techniques are coupled by eager propagation methods, then the concurrency control methods can adequately address the concurrent updates problem. However, if lazy propagation methods are used, then transactions may be executed in different orders at different sites causing non-1SR global history. Furthermore, various replicas will get out of sync. To manage these problems, a reconciliation method is aplied involving undoing and redoing transactions in such a way that transaction execution is the same at each site. This is not an easy issue since the reconciliation is generally application dependent.

13.3 REPLICATION PROTOCOLS

In the previous section, we discussed two dimensions along which update management techniques can be classified. These dimensions are orthogonal; therefore four combinations are possible: eager centralized, eager distributed, lazy centralized, and lazy distributed. We discuss each of these alternatives in this section. For simplicity of exposition, we assume a fully replicated database, which means that all update transactions are global. We further assume that each site implements a 2PL-based concurrency control technique.

13.3.1 Eager Centralized Protocols

In eager centralized replica control, a master site controls the operations on a data item. These protocols are coupled with strong consistency techniques, so that updates are applied to all the replicas within the context of the update transaction, which is committed using the 2PC protocol (although non-2PC alternatives exist as we discuss shortly). Consequently, once the update transaction completes, all replicas have the same values for the updated data items, and the resulting global history is 1SR.

There are two design parameters that determine the specific implementation of eager centralized replica control protocols: distribution of masters among data items, and the level of replication transparency that is provided. We discussed the first parameter earlier, in Section 13.2.3. To repeat, it refers to whether there is a single master site for all data items (single master), or different master sites for each, or, more likely, for a group of data items (primary copy). The second parameter refers to whether or not the user applications need to know the master site.

The simplest case is to have a single master for the entire database (i.e., for all data items) with limited replication transparency so that user applications know the master site. In this case, global update transactions (i.e., those which contain at least one $Write(x)$ operation where x is a replicated data item) are submitted directly to the master site – more specifically, to the transaction manager (TM) at the master site. At the master, each $Read(x)$ operation is performed on the master copy (i.e., $Read(x)$ is converted to $Read(x_M)$, where M signifies master copy) and executed as follows: a read lock is obtained on x_M, the read is performed, and the result is returned to the user. Similarly, each $Write(x)$ causes an update of the master copy (i.e., executed as $Write(x_M)$) by first obtaining a write lock and then performing the write operation. The master TM then forwards the $Write$ to the slave sites either synchronously or in a deferred fashion (Figure 13.1). In either case, it is important to propagate updates such that conflicting updates are executed at the slaves in the same order they are executed at the master.. This can be achieved by timestamping or by some other ordering scheme.

If the user application is submitting a read transaction (i.e., all operations are $Read$), then it can submit it to any slave site. The execution of read transactions at the slaves can follow the process of centralized concurrency control algorithms, such as C2PL (Algorithms 11.1-11.3), where the centralized lock manager site also acts as the master replica site. Implementations within C2PL require minimal changes to the TM at the non-master sites, primarily to deal with the $Write$ operations as described above, and its consequences (e.g., in the processing of Commit command). Thus, when a slave site receives a $Read$ op-

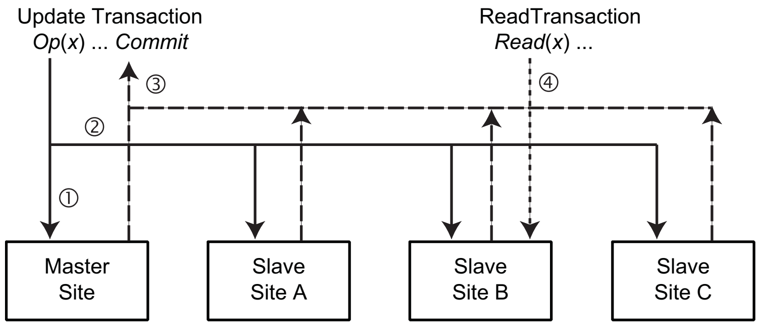

Figure 13.1: Eager Single Master Replication Protocol Actions. (1) A $Write$ is applied on the master copy; (2) $Write$ is then propagated to the other replicas; (3) Updates become permanent at commit time; (4) A $Read$ goes to any slave copy.

eration, it forwards it to the master site to obtain a read lock. The $Read$ can then be executed at the master and the result returned to the application, or the master can simply send a "lock granted" message to the originating site, which can then execute the $Read$ on the local copy.

However, this puts too much load on the master. It is possible to perform the $Read$ on the local copy without obtaining a read lock from the master site. Whether synchronous or deferred propagation is used, the local concurrency control algorithm ensures that the local read-write conflicts are properly serialized, and since the $Write$ operations can only be coming from the master as part of update propagation, local write-write conflicts won't occur as the propagation transactions are executed in each slave in the order dictated by the master. However, a $Read$ may read data item values at a slave either before an update is installed or after. The fact that a read transaction at one slave site may read the value of one replica before an update while another read transaction reads another replica at another slave after the same update is inconsequential from the perspective of ensuring global 1SR histories. This is demonstrated by the following example.

Example 13.3

Consider a data item x whose master site is at Site A with slaves at sites B and C. Consider the following three transactions:

T_1:	Write(x)	T_2:	Read(x)	T_3:	Read(x)
	Commit		Commit		Commit

Assume that T_2 is sent to slave at Site B and T_3 to slave at Site C. Assume that T_2 reads x at B [$Read(x_B)$] before T_1's update is applied at B, while T_3 reads x at C [$Read(x_C)$] after T_1's update at C. Then the histories generated at the two slaves will be as follows:

$$\begin{aligned} H_B &= \{R_2(x), C_2, W_1(x), C_1\} \\ H_C &= \{W_1(x), C_1, R_3(x), C_3\} \end{aligned}$$

The serialization order at Site B is $T_2 \rightarrow T_1$, while at Site C it is $T_1 \rightarrow T_3$. The global serialization order, therefore, is $T_2 \rightarrow T_1 \rightarrow T_3$, which is fine. Therefore the history is 1SR.

Consequently, if this approach is followed, read transactions may read data that are concurrently updated at the master, but the global history will still be 1SR.

In this alternative protocol, when a slave site receives a $Read(x)$, it reads from its local copy (i.e., $Read(x_i)$) and returns the result to the user application; this can only come from a read transaction. When it receives a $Write(x)$, if the $Write$ is coming from the master site, then it performs it on the local copy (i.e., $Write(x_i)$). If it receives a $Write$ from a user application, then it rejects it, since this is obviously an error given that update transactions have to be submitted to the master site.

These alternatives of a single master eager centralized protocol are simple to implement. One important issue to address is how one recognizes a transaction as "update" or "read-only" – it may be possible to do this by explicit declaration within the `Begin_Transaction` command. However, they require each user application to know the master site, and they both put significant load on the master that has to deal with (at least) the $Read$ operations within update transactions as well as acting as the coordinator for these transactions during 2PC execution.

These issues can be addressed, to some extent, by involving, in the execution of the update transactions, the TM at the site where the application runs. Thus the update transactions are not submitted to the master, but to the TM at the site where the application runs. This TM can act as the coordinating TM for both update and read-only transactions. Applications can simply submit their read as well as update transactions to their local TM, providing full transparency.

There are alternatives to implementing full transparency – the coordinating TM may only act as a "router", forwarding each operation directly to the master site. The master site can then execute the operations locally (as described above) and return the results to the application. Although this alternative implementation provides full transparency and has the advantage of being simple to implement, it does not address the overloading problem at the master. An alternative implementation may be as follows.

1. The coordinating TM sends each operation, as it gets it, to the central (master) site. This requires no change to the C2PL-TM algorithm (Algorithm 11.1).
2. If the operation is a $Read(x)$, then the centralized lock manager (C2PL-LM in Algorithm 11.2) can proceed by setting a read lock on its copy of x (call it x_M) on behalf of this transaction and informs the coordinating TM that the read lock is granted. The coordinating TM can then forward the $Read(x)$ to any slave site that holds a replica of x (i.e., converts it to a $Read(x_i)$). The read can then be carried out by the data processor (DP) at that slave.
3. If the operation is a $Write(x)$, then the centralized lock manager (master) proceeds as follows:
   - **(a)** It first sets a write lock on its copy of x.

(b) It then calls its local DP to perform the $Write$ on its own copy of x (i.e., converts the operation to $Write(x_M)$).

(c) Finally, it informs the coordinating TM that the write lock is granted.

The coordinating TM, in this case, sends the $Write(x)$ to all the slaves where a copy of x exists; the DPs at these slaves apply the $Write$ to their local copies.

The fundamental difference in this case is that the master site does not deal with $Read$s or with the coordination of the updates across replicas. These are left for the TM at the site where the user application runs.

It is straightforward to see that this algorithm guarantees that the histories are 1SR since the serialization orders are determined at a single master (similar to centralized concurrency control algorithms). It is also clear that it follows the ROWA protocol, as discussed above – since all the copies are ensured to be up-to-date when an update transaction completes, a $Read$ can be performed on any copy.

To demonstrate how eager algorithms combine replica control and concurrency control, we show how the Transaction Management algorithm for the coordinating TM (Algorithm 13.1) and the Lock Management algorithm for the master site (Algorithm 13.2). We show only the revisions to the centralized 2PL algorithms (Algorithms 11.1 and 11.2 in Chapter 11).

```
begin
    ⋮
    if lock request granted then
        if op.Type = W then
            $S \leftarrow$ set of all sites that are slaves for the data item
        else
            $S \leftarrow$ any one site which has a copy of data item
        endif
        DP$_S(op)$                         {send operation to all sites in set $S$}
    else
        inform user about the termination of transaction
    endif
    ⋮
end
```

Algorithm 13.1: Eager Single Master Modifications to C2PL-TM

Note that in the algorithm fragments that we have given, the LM simply sends back a "Lock granted" message and not the result of the update operation. Consequently, when the update is forwarded to the slaves by the coordinating TM, they need to execute the update operation themselves. This is sometimes referred to as *operation transfer*. The alternative is for the "Lock granted" message to include the result of the update computation, which is then forwarded to the slaves who simply need to apply the result and update their logs. This is referred to as *state transfer*. The distinction may seem trivial if the operations are simply

```
begin
  ⋮
  switch op.Type do
    case R or W                    {lock request; see if it can be granted}
      find the lock unit $lu$ such that $op.arg \subseteq lu$ ;
      if $lu$ is unlocked or lock mode of $lu$ is compatible with $op.Type$ then
        set lock on $lu$ in appropriate mode on behalf of transaction $op.tid$ ;
        if op.Type = W then
          $DP_M(op)$          {call local DP (M for "master") with operation}
        send "Lock granted" to coordinating TM of transaction
      else
        put $op$ on a queue for $lu$
      endif
    endsw
    ⋮
  endsw
end
```

Algorithm 13.2: Eager Single Master Modifications to C2PL-LM

in the form $Write(x)$, but recall that this $Write$ operation is an abstraction; each update operation may require the execution of an SQL expression, in which case the distinction is quite important.

The above implementation of the protocol relieves some of the load on the master site and alleviates the need for user applications to know the master. However, its implementation is more complicated than the first alternative we discussed. In particular, now the TM at the site where transactions are submitted has to act as the 2PC coordinator and the master site becomes a participant. This requires some care in revising the algorithms at these sites.

Let us now relax the requirement that there is one master for all data items; each data item can have a different master. In this case, for each replicated data item, one of the replicas is designated as the *primary copy*. Consequently, there is no single master to determine the global serialization order, so more care is required. First note that, in this case, limited replication transparency option only makes sense if an update transaction accesses only data items whose primary sites are at the same site. Otherwise, the application program cannot forward the update transactions to one master; it will have to do it operation-by-operation, and, furthermore, it is not clear which primary copy master would serve as the coordinator for 2PC execution. Therefore, the reasonable alternative is the full replication transparency support, where the TM at the application site acts as the coordinating TM and forwards each operation to the primary site of the data item that it acts on. Figure 13.2 depicts the sequence of operations in this case. Site A is the master for data item x and sites B and C hold replicas (i.e., they are slaves); similarly data item y's master is site C with slave sites B and D.

Recall that this version still applies the updates to all the replicas within transactional

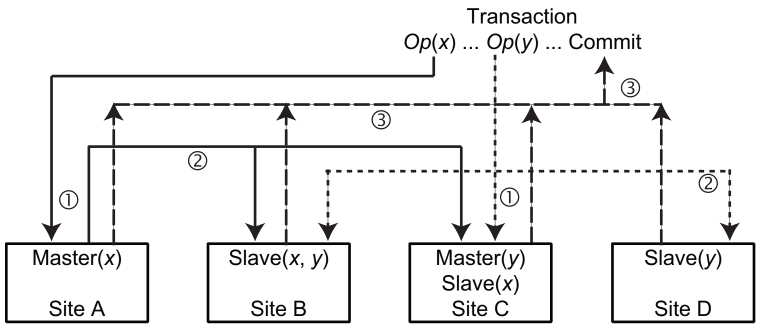

Figure 13.2: Eager Primary Copy Replication Protocol Actions. (1) Operations ($Read$ or $Write$) for each data item are routed to that data item's master and a $Write$ is first applied at the master; (2) $Write$ is then propagated to the other replicas; (3) Updates become permanent at commit time.

boundaries, requiring integration with concurrency control techniques. A very early proposal is the *primary copy two-phase locking* (PC2PL) algorithm proposed for the prototype distributed version of INGRES [Stonebraker and Neuhold, 1977]. PC2PL is a straightforward extension of the single master protocol discussed above in an attempt to counter the latter's potential performance problems. Basically, it implements lock managers at a number of sites and makes each lock manager responsible for managing the locks for a given set of lock units for which it is the master site. The transaction managers then send their lock and unlock requests to the lock managers that are responsible for that specific lock unit. Thus the algorithm treats one copy of each data item as its primary copy.

As a combined replica control/concurrency control technique, primary copy approach demands a more sophisticated directory at each site, but it also improves the previously discussed approaches by reducing the load of the master site without causing a large amount of communication among the transaction managers and lock managers.

13.3.2 Eager Distributed Protocols

In eager distributed replica control, the updates can originate anywhere, and they are first applied on the local replica[3], then the updates are propagated to other replicas. In this case, the local replica is the master for that update, while the other replicas are slaves. Again, all of these are done within the context of the update transaction, and when the transaction commits, the user is notified and the updates are made permanent. Figure 13.3 depicts the sequence of operations for one logical data item x with copies at sites A, B, and D, and where two transactions update two different copies (at sites A and D).

As can be clearly seen, the critical issue is to ensure that concurrent conflicting $Write$s initiated at different master sites are executed in the same order at each site are properly

[3]We are assuming that the update originates at a site where a replica of the data item exists; otherwise it is forwarded to any of the replica sites.

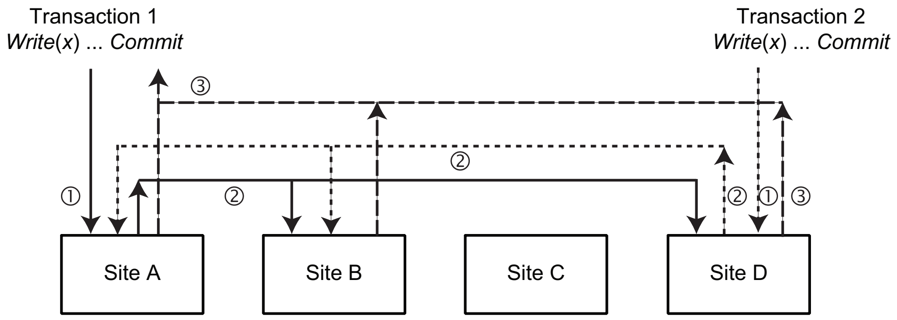

Figure 13.3: Eager Distributed Replication Protocol Actions. (1) Two $Write$ operations are applied on two local replicas of the same data item; (2) The $Write$ operations are independently propagated to the other replicas; (3) Updates become permanent at commit time (shown only for Transaction 1).

synchronized (of course, the local executions at each site also have to be serializable). This is achieved by means of the concurrency control techniques that are employed at each site. Consequently, read operations can be performed on any copy, but writes are performed on all copies within transactional boundaries (e.g., ROWA) using a concurrency control protocol.

13.3.3 Lazy Centralized Protocols

Lazy centralized replication algorithms are similar to eager centralized replication ones in that the updates are first applied to a master replica and then propagated to the slaves. The important difference is that the propagation does not take place within the update transaction, but after the transaction commits as a separate refresh transaction. Consequently, if a slave site performs a $Read(x)$ operation on its local copy, it may read stale (non-fresh) data, since x may have been updated at the master, but the update may not have yet been propagated to the slaves.

In the case of single master with limited transparency, the update transactions are submitted and executed directly at the master site (as in the eager single master); once the update transaction commits, then the refresh transaction is sent to the slaves. The sequence of execution steps are as follows: (1) an update transaction is first applied to the master replica, (2) the transaction is committed at the master, and then (3) the refresh transaction is sent to the slaves (Figure 13.4).

When a slave (secondary) site receives a $Read(x)$, it reads from its local copy and returns the result to the user. Notice that, as indicated above, its own copy may not be up-to-date if the master is being updated and the slave has not yet received and executed the corresponding refresh transaction. A $Write(x)$ received by a slave is rejected (and the transaction aborted), as this should have been submitted directly to the master site. When a slave receives a refresh transaction from the master, it applies the updates to its local copy. When it receives a $Commit$ or $Abort$ (this can happen for only locally submitted

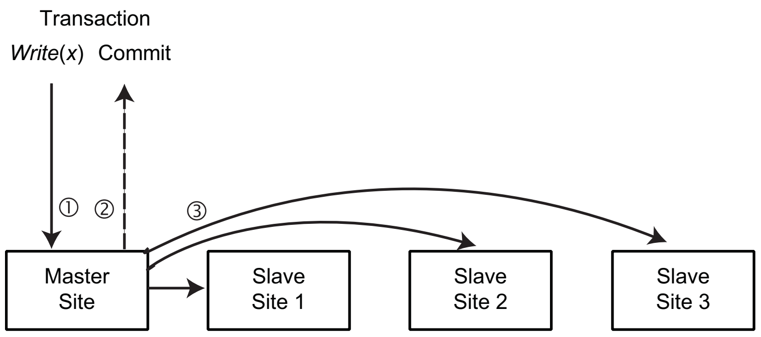

Figure 13.4: Lazy Single Master Replication Protocol Actions. (1) Update is applied on the local replica; (2) Transaction commit makes the updates permanent at the master; (3) Update is propagated to the other replicas in refresh transactions; (4) Transaction 2 reads from local copy.

read-only transactions), it locally performs these actions.

How can it be ensured that the refresh transactions can be applied at all of the slaves in the same order? In this architecture, since there is a single master site that stores all master copies, the ordering can be established by simply using timestamps. The master site would attach a timestamp to each refresh transaction according to the commit order of the actual update transaction, and the slaves would apply the refresh transactions in timestamp order.

A similar approach may be followed in the primary copy, limited transparency case (i.e., update transactions are submitted at a master, which may be different for different data items). In this case, a site may contain slave copies of a number of data items, causing it to get refresh transactions from multiple masters. The execution of these refresh transactions need to be ordered the same way at all of the involved slaves to ensure that the database states eventually converge. There are a number of alternatives that can be followed.

One alternative is to assign timestamps such that refresh transactions issued from different masters have different timestamps (by appending the site identifier to a monotonic counter at each site). Then the refresh transactions at each site can be executed in their timestamp order. However, those that come out of order cause difficulty. In traditional timestamp-based techniques discussed in Chapter 11, these transactions would be aborted; however in a lazy technique, this is not possible since the transaction has already been committed at the master site. The only possibility is to run a compensating transaction (which, effectively, aborts the transaction by rolling back its effects) or to perform update reconciliation that will be discussed shortly. The issue can be addressed by a more careful study of the resulting histories. An approach proposed by Breitbart and Korth [1997] uses a serialization graph approach that builds a *replication graph* whose nodes consist of transactions (T) and sites (S) and an edge $\langle T_i, S_j \rangle$ exists in the graph if and only if T_i performs a $Write$ on a replicated data item that is stored at S_j. When an operation (op_k) is submitted, the appropriate nodes (T_k) and edges are inserted into the replication graph, which is checked for cycles. If there is no cycle, then the execution can proceed. If a cycle is detected and it involves a transaction that has committed at the master, but whose refresh transactions have not yet committed at all of the involved slaves, then the current transaction (T_k) is aborted (to be restarted later) since its execution would cause the history to be non-1SR. If the cycle is formed, but none of the transactions involved in the cycle have yet committed at the master, then T_k can wait until the other transactions in the cycle are completed (i.e., they are committed at their masters and their refresh transactions are committed at all of the slaves). When a transaction is completed in this manner, the corresponding node and all of its incident edges are removed from the replication graph. This protocol is proven to produce 1SR histories. An important issue is the maintenance of the replication graph. If it is maintained by a single site, then this becomes a centralized algorithm. We leave the distributed construction and maintenance of the replication graph as an exercise.

Another alternative is to rely on the group communication mechanism provided by the underlying communication infrastructure (if it can provide it). We discuss this alternative in Section 13.4.

Recall that in Section 13.3.1 we indicated that eager primary copy, limited replication approach makes sense if the update transactions access only data items whose master sites are the same, since the update transactions are run completely at a master. The same problem exists in the case of lazy primary copy, limited replication approach. The issue that arises in both cases is how to design the distributed database so that meaningful transactions can be executed. This problem has been studied within the context of lazy protocols [Chundi et al., 1996] and a primary site selection algorithm was proposed that, given a set of transactions, a set of sites, and a set of data items, finds a primary site assignment to these data items (if one exists) such that the set of transactions can be executed to produce a 1SR global history.

We now turn to alternatives that provide full transparency by allowing (both read and update) transactions to be submitted at any site and forwarding their operations to either the single master or to the appropriate primary master site. This is tricky and involves two problems: the first is that, unless one is careful, 1SR global history may not be guaranteed; the second problem is that a transaction may not see its own updates. The following two examples demonstrate these problems.

Example 13.4

Consider the single master scenario and two sites M and B where M holds the master copies of x and y and B holds their slave copies. Now consider the following two transactions: T_1 submitted at site B, while transaction T_2 submitted at site M:

T_1:	Read(x)	T_2:	Write(x)
	Write(y)		Write(y)
	Commit		Commit

One way these would be executed under full transparency is as follows. T_2 would be executed at site M since it contains the master copies of both x and y. Sometime after it commits, refresh transactions for its $Write$s are sent to site B to update the slave copies. On the other hand, T_1 would read the local copy of x at site B, but its $Write(x)$ would be forwarded to x's master copy, which is at site M. Some time after $Write_1(x)$ is executed at the master site and commits there, a refresh transaction would be sent back to site B to update the slave copy. The following is a possible sequence of steps of execution (Figure 13.5):

1. $Read_1(x)$ is submitted at site B, where it is performed;
2. $Write_2(x)$ is submitted at site M, and it is executed;
3. $Write_2(y)$ is submitted at site M, and it is executed;
4. T_2 submits its $Commit$ at site M and commits there;
5. $Write_1(x)$ is submitted at site B; since the master copy of x is at site M, the $Write$ is forwarded to M;
6. $Write_1(x)$ is executed at site M and the confirmation is sent back to site B;
7. T_1 submits $Commit$ at site B, which forwards it to site M; it is executed there and B is informed of the commit where T_1 also commits;
8. Site M now sends refresh transaction for T_2 to site B where it is executed and commits;
9. Site M finally sends refresh transaction for T_1 to site B (this is for T_1's $Write$ that was executed at the master), it is executed at B and commits.

The following two histories are now generated at the two sites where the superscript R on operations indicate that they are part of a refresh transaction:

$$\begin{aligned} H_M &= \{W_2(x_M), W_2(y_M), C_2, W_1(y_M), C_1\} \\ H_B &= \{R_1(x_B), C_1, W_2^R(x_B), W_2^R(y_B), C_2^R, W_1^R(x_B), C_1^R\} \end{aligned}$$

The resulting global history over the *logical* data items x and y is non-1SR.

Example 13.5

Again consider a single master scenario, where site M holds the master copy of x and site C holds its slave. Consider the following simple transaction:

T_3: Write(x)
Read(x)
Commit

Following the same execution model as in Example 13.4, the sequence of steps would be as follows:

1. $Write_3(x)$ is submitted at site C, which forwards it to site M for execution;
2. The $Write$ is executed at M and the confirmation is sent back to site B;
3. $Read_3(x)$ is submitted at site C and is executed on the local copy;

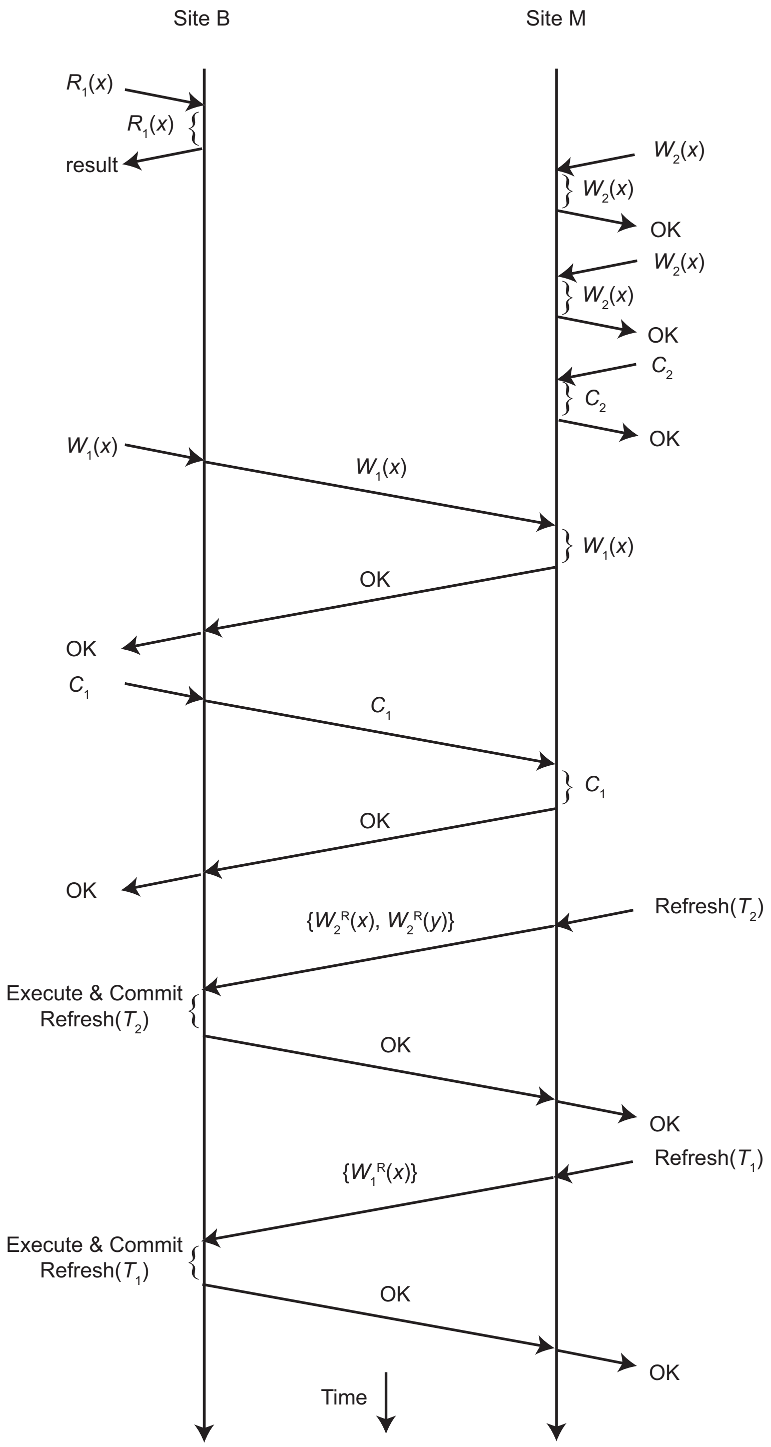

Figure 13.5: Time sequence of executions of transactions

4. T_3 submits commit at C, which is forwarded to M, executed there and a notification is sent back to C, which also commits the transaction;
5. Site M sends a refresh transaction to C for the $W_3(x)$ operation;
6. Site C executes the refresh transaction and commits it.

Note that, since the refresh transaction is sent to C sometime after T_3 commits at site M, at step 3 when it reads the value of x at site C, it reads the old value and does not see the value of its own $Write$ that just precedes $Read$.

Because of these problems, there are not too many proposals for full transparency in lazy replication algorithms. A notable exception is that by Bernstein et al. [2006] that considers the single master case and provides a method for validity testing by the master site, at commit point, similar to optimistic concurrency control. The fundamental idea is the following. Consider a transaction T that writes a data item x. At commit time of transaction T, the master generates a timestamp for it and uses this timestamp to set a timestamp for the master copy of x (x_M) that records the timestamp of the last transaction that updated it ($last\_modified(x_M)$). This is appended to refresh transactions as well. When refresh transactions are received at slaves they also set their copies to this same value, i.e., $last\_modified(x_i) \leftarrow last\_modified(x_M)$. The timestamp generation for T at the master follows the following rule:

The timestamp for transaction T should be greater than all previously issued timestamps and should be less than the $last\_modified$ timestamps of the data items it has accessed. If such a timestamp cannot be generated, then T is aborted.[4]

This test ensures that read operations read correct values. For example, in Example 13.4, master site M would not be able to assign an appropriate timestamp to transaction T_1 when it commits, since the $last\_modified(x_M)$ would reflect the update performed by T_2. Therefore, T_1 would be aborted.

Although this algorithm handles the first problem we discussed above, it does not automatically handle the problem of a transaction not seeing its own writes (what we referred to as transaction inversion earlier). To address this issue, it has been suggested that a list be maintained of all the updates that a transaction performs and this list is consulted when a $Read$ is executed. However, since only the master knows the updates, the list has to be maintained at the master and all the $Read$s (as well as $Write$s) have to be executed at the master.

13.3.4 Lazy Distributed Protocols

Lazy distributed replication protocols are the most complex ones owing to the fact that updates can occur on any replica and they are propagated to the other replicas lazily (Figure 13.6).

[4]The original proposal handles a wide range of freshness constraints, as we discussed earlier; therefore, the rule is specified more generically. However, since our discussion primarily focuses on 1SR behavior, this (more strict) recasting of the rule is appropriate.

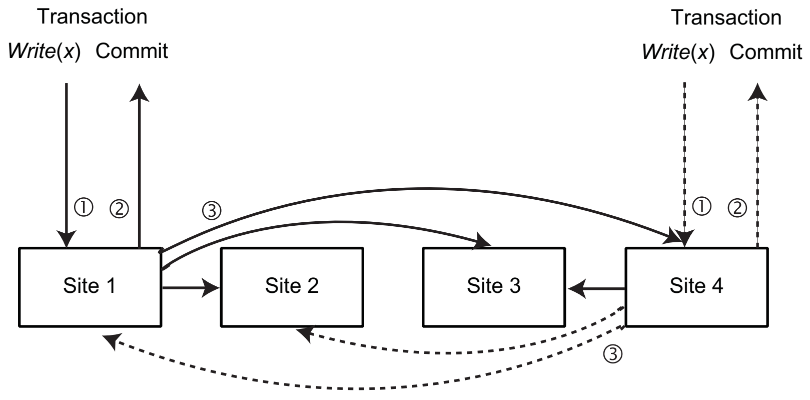

Figure 13.6: Lazy Distributed Replication Protocol Actions. (1) Two updates are applied on two local replicas; (2) Transaction commit makes the updates permanent; (3) The updates are independently propagated to the other replicas.

The operation of the protocol at the site where the transaction is submitted is straightforward: both $Read$ and $Write$ operations are executed on the local copy, and the transaction commits locally. Sometime after the commit, the updates are propagated to the other sites by means of refresh transactions.

The complications arise in processing these updates at the other sites. When the refresh transactions arrive at a site, they need to be locally scheduled, which is done by the local concurrency control mechanism. The proper serialization of these refresh transactions can be achieved using the techniques discussed in previous sections. However, multiple transactions can update different copies of the same data item concurrently at different sites, and these updates may conflict with each other. These changes need to be reconciled, and this complicates the ordering of refresh transactions. Based on the results of reconciliation, the order of execution of the refresh transactions is determined and updates are applied at each site.

The critical issue here is reconciliation. One can design a general purpose reconciliation algorithm based on heuristics. For example, updates can be applied in timestamp order (i.e., those with later timestamps will always win) or one can give preference to updates that originate at certain sites (perhaps there are more important sites). However, these are ad hoc methods and reconciliation is really dependent upon application semantics. Note that timestamp-based ordering will only work if timestamps are based on local clocks that are synchronized. As we discussed earlier, this is hard to achieve in large-scale distributed systems. Simple timestamps based on concatenation of a site number and local clock give arbitrary preference between transactions that may have no real basis in application logic. The reason timestamps work well in concurrency control and not in this case is because in concurrency control we are only interested in determining *some* order; here we are interested in determining a *particular* order that is consistent with application semantics. Furthermore, whatever reconciliation technique is used, some of the updates are lost.

13.4 GROUP COMMUNICATION

A very simple cost model for the replication algorithms is as follows. If there are n replicas and each transaction consists of m update operations, then each transaction issues $n * m$ messages (if multicast communication is possible, m messages would be sufficient). If the system wishes to maintain a throughput of k transactions-per-second, this results in $k * n * m$ messages per second (or $k * m$ in the case of multicasting). One can add sophistication to this cost function by considering the execution time of each operation (perhaps based on system load) to get a cost function in terms of time. The problem with many of the replication protocols discussed above (in particular the distributed ones) is that their message overhead is significant.

A critical issue in efficient implementation of these protocols is to reduce the message overhead. Solutions have been proposed that use group communication protocols [Chockler et al., 2001] together with non-traditional local transaction processing [Stanoi et al., 1998; Kemme and Alonso, 2000a,b; Patiño-Martínez et al., 2000; Jiménez-Peris et al., 2002]. These solutions introduce two modifications: they do not employ 2PC at commit time, but rely on the underlying group communication protocols to ensure agreement, and they use deferred update propagation rather than synchronous.

Let us first review the group communication idea. A group communication system enables a node to multicast a message to all nodes of a group with a delivery guarantee, i.e. the message is eventually delivered to all nodes. Furthermore, it can provide multicast primitives with different delivery orders only one of which is important for our discussion: total order. In total ordered multicast, all messages sent by different nodes are delivered in the same total order at all nodes. This is important in understanding the following discussion.

We will demonstrate the use of group communication by considering two protocols. The first one [Kemme and Alonso, 2000a] is an alternative eager distributed protocol, while the second one is a lazy centralized protocol [Pacitti et al., 1999].

The group communication-based eager distributed protocol due to Kemme and Alonso [2000a] uses a local processing strategy where $Write$ operations are carried out on local shadow copies where the transaction is submitted and utilizes total ordered group communication to multicast the set of write operations of the transaction to all the other replica sites. Total ordered communication guarantees that all sites receive the write operations in exactly the same order, thereby ensuring identical serialization order at every site. For simplicity of exposition, in the following discussion, we assume that the database is fully replicated and that each site implements a 2PL concurrency control algorithm.

The protocol executes a transaction T_i in four steps (local concurrency control actions are not indicated):

I. **Local processing phase.** A $Read_i(x)$ operation is performed at the site where it is submitted (this is the master site for this transaction). A $Write_i(x)$ operation is also performed at the master site, but on a shadow copy (see last chapter for a discussion of shadow paging).

II. **Communication phase.** If T_i consists only of $Read$ operations, then it can be committed at the master site. If it involves $Write$ operations (i.e., if it is an update transaction), then the TM at T_i's master site (i.e., the site where T_i is submitted) assembles the writes into one *write message* WM_i[5] and multicasts it to all the sites (including itself) using total ordered group communication.

III. **Lock phase.** When WM_i is delivered at a site S_j, Site S_j requests all locks in WM_i in an atomic step. This can be done by acquiring a latch (lighter form of a lock) on the lock table that is kept until all the locks are granted or requests are enqueued. The following actions are performed:

   1. For each $Write(x)$ in WM_i (let x_j refer to the copy of x that exists at site S_j), the following are performed:

      (a) If there are no other transactions that have locked x_j, then the write lock on x_j is granted.

      (b) Otherwise a conflict test is performed:

      - If there is a local transaction T_k that has already locked x_j, but is in its local read or communication phases, then T_k is aborted. Furthermore, if T_k is in its communication phase, a final decision message $Abort$ is multicast to all the sites. At this stage, read/write conflicts are detected and local read transactions are simply aborted. Note that only local read operations obtain locks during the local execution phase, since local writes are only executed on shadow copies. Therefore, there is no need to check for write/write conflicts at this stage.
      - Otherwise, $W_i(x_j)$ lock request is put on queue for x_j.

   2. If T_i is a local transaction (recall that the message is also sent to the site where T_i originates, so, in this case $j = i$), then the site can commit the transaction, so it multicasts a $Commit$ message. Note that the commit message is sent as soon as the locks are requested and not after Writes; thus this is not a 2PC execution.

IV. **Write phase.** When a site is able to obtain the write lock, it applies the corresponding update (for the master site, this means that the shadow copy is made the valid version). The site where T_i is submitted can commit and release all the locks. Other sites have to wait for the decision message and terminate accordingly.

Note that in this protocol, the important thing is to ensure that the lock phases of the concurrent transactions are executed in the same order at each site; that is what total ordered multicasting achieves. Also note that there is no ordering requirement on the decision messages (step III.2) and these may be delivered in any order, even before the delivery of

[5] What is being sent are the updated data items (i.e., state transfer).

the corresponding WM. If this happens, then the sites that receive the decision message before WM simply register the decision, but do not take any action. When WM message arrives, they can execute the lock and write phases and terminate the transaction according to the previously delivered decision message.

This protocol is significantly better, in terms of performance, than the naive one discussed in Section 13.3.2. For each transaction, the master site sends two messages: one when it sends the WM and the second one when it communicates the decision. Thus, if we wish to maintain a system throughput of k transactions-per-second, the total number of messages is $2k$ rather than $k * m$, which is the case with the naive protocol (assuming multicast in both cases). Furthermore, system performance is improved by the use of deferred eager propagation since synchronization among replica sites for all $Write$ operations is done once at the end rather than throughout the transaction execution.

The second example of the use of group communication that we will discuss is in the context of lazy centralized algorithms. Recall that an important issue in this case is to ensure that the refresh transactions are ordered the same way at all the involved slaves so that the database states converge. If totally ordered multicasting is available, the refresh transactions sent by different master sites would be delivered in the same order at all the slaves. However, total order multicast has high messaging overhead which may limit its scalability. It is possible to relax the ordering requirement of the communication system and let the replication protocol take responsibility for ordering the execution of refresh transactions. We will demonstrate this alternative by means of a proposal due to Pacitti et al. [1999]. The protocol assumes FIFO ordered multicast communication with a bounded delay for communication (call it Max), and assumes that the clocks are loosely synchronized so that they may only be out of sync by up to ϵ. It further assumes that there is an appropriate transaction management functionality at each site. The result of the replication protocol at each slave is to maintain a "running queue" that holds an ordered list of refresh transactions, which is the input to the transaction manager for local execution. Thus, the protocol ensures that the orders in the running queues at each slave site where a set of refresh transactions run are the same.

At each slave site, a "pending queue" is maintained for each master site of this slave (i.e., if the slave site has replicas of x and y whose master sites are $Site_1$ and $Site_2$, respectively, then there are two pending queues, q_1 and q_2, corresponding to master sites $Site_1$ and $Site_2$, respectively). When a refresh transaction RT_i^k is created at a master site $Site_k$, it is assigned a timestamp $ts(RT_i)$ that corresponds to the real time value at the commitment time of the corresponding update transaction T_i. When RT_i arrives at a slave, it is put on queue q_k. At each message arrival the top elements of all pending queues are scanned and the one with the lowest timestamp is chosen as the new RT (new_RT) to be handled. If the new_RT has changed since the last cycle (i.e., a new RT arrived with a lower timestamp than what was chosen in the previous cycle), then the one with the lower timestamp becomes the new_RT and is considered for scheduling.

When a refresh transaction is chosen as the new_RT, it is not immediately put on the "running queue" for the transaction manager; the scheduling of a refresh transaction takes into account the maximum delay and the possible drift in local clocks. This is done to ensure that any refresh transaction that may be delayed has a chance of reaching the slave. The time when an RT_i is put into the "running queue" at a slave site is

$delivery\_time = ts(new\_RT) + Max + \epsilon$. Since the communication system guarantees an upper bound of Max for message delivery and since the maximum drift in local clocks (that determine timestamps) is ϵ, a refresh transaction cannot be delayed by more than the $delivery\_time$ before reaching all of the intended slaves. Thus, the protocol guarantees that a refresh transaction is scheduled for execution at a slave when the following hold: (1) all the write operations of the corresponding update transaction are performed at the master, (2) according to the order determined by the timestamp of the refresh transaction (which reflects the commit order of the update transaction), and (3) at the earliest at real time equivalent to its $delivery\_time$. This ensures that the updates on secondary copies at the slave sites follow the same chronological order in which their primary copies were updated and this order will be the same at all of the involved slaves, assuming that the underlying communication infrastructure can guarantee Max and ϵ. This is an example of a lazy algorithm that ensures 1SR global history, but weak mutual consistency, allowing the replica values to diverge by up to a predetermined time period.

13.5 REPLICATION AND FAILURES

Up to this point, we have focused on replication protocols in the absence of any failures. What happens to mutual consistency concerns if there are system failures? The handling of failures differs between eager replication and lazy replication approaches.

13.5.1 Failures and Lazy Replication

Let us first consider how lazy replication techniques deal with failures. This case is relatively easy since these protocols allow divergence between the master copies and the replicas. Consequently, when communication failures make one or more sites unreachable (the latter due to network partitioning), the sites that are available can simply continue processing. Even in the case of network partitioning, one can allow operations to proceed in multiple partitions independently and then worry about the convergence of the database states upon repair using the conflict resolution techniques discussed in Section 13.3.4. Before the merge, databases at multiple partitions diverge, but they are reconciled at merge time.

13.5.2 Failures and Eager Replication

Let us now focus on eager replication, which is considerably more involved. As we noted earlier, all eager techniques implement some sort of ROWA protocol, ensuring that, when the update transaction commits, all of the replicas have the same value. ROWA family of protocols is attractive and elegant. However, as we saw during the discussion of commit protocols, it has one significant drawback. Even if one of the replicas is unavailable, then the update transaction cannot be terminated. So, ROWA fails in meeting one of the fundamental goals of replication, namely providing higher availability.

An alternative to ROWA which attempts to address the low availability problem is the Read One Write All Available (ROWA-A) protocol. The general idea is that the write commands are executed on all the available copies and the transaction terminates. The copies that were unavailable at the time will have to "catch up" when they become available.

There have been various versions of this protocol [Helal et al., 1997], two of which will be discussed here. The first one is known as the *available copies protocol* [Bernstein and Goodman, 1984; Bernstein et al., 1987]. The coordinator of an update transaction T_i (i.e., the master where the transaction is executing) sends each $W_i(x)$ to all the slave sites where replicas of x reside, and waits for confirmation of execution (or rejection). If it times out before it gets acknowledgement from all the sites, it considers those which have not replied as unavailable and continues with the update on the available sites. The unavailable slave sites, of course, update their databases to the latest state when they recover. Note, however, that these sites may not even be aware of the existence of T_i and the update to x that T_i has made if they had become unavailable before T_i started.

There are two complications that need to be addressed. The first one is the possibility that the sites that the coordinator thought were unavailable were in fact up and running and may have already updated x but their acknowledgement may not have reached the coordinator before its timer ran out. Second, some of these sites may have been unavailable when T_i started and may have recovered since then and have started executing transactions. Therefore, the coordinator undertakes a validation procedure before committing:

1. The coordinator checks to see if all the sites it thought were unavailable are still unavailable. It does this by sending an inquiry message to every one of these sites. Those that are available reply. If the coordinator gets a reply from one of these sites, it aborts T_i since it does not know the state that the previously unavailable site is in: it could have been that the site was available all along and had performed the original $W_i(x)$ but its acknowledgement was delayed (in which case everything is fine), or it could be that it was indeed unavailable when T_i started but became available later on and perhaps even executed $W_j(x)$ on behalf of another transaction T_j. In the latter case, continuing with T_i would make the execution schedule nonserializable.

2. If the coordinator of T does not get any response from any of the sites that it thought were unavailable, then it checks to make sure that all the sites that were available when $W_T(x)$ executed are still available. If they are, then T can proceed to commit. Naturally, this second step can be integrated into a commit protocol.

The second ROWA-A variant that we will discuss is the distributed ROWA-A protocol. In this case each site maintains a set, V, of sites that it believes is available; for a site S, $V(S)$ is the "view" that S has of the system configuration. In particular, when a transaction T_i is submitted, its coordinator's view reflects all the sites that the coordinator knows to be available (let us denote this as $V(T_i)$ for simplicity. A $R_i(x)$ is performed on any replica in $V(T_i)$ and a $W_i(x)$ updates all copies in $V(T_i)$. The coordinator checks its view at the end of T_i, and if the view has changed since T_i's start, then T_i is aborted. To modify V, a special atomic transaction is run at all sites, ensuring that no concurrent views are generated. This can be achieved by assigning timestamps to each V when it is generated and ensuring that a site only accepts a new view if its version number is greater than the version number of that site's current view.

The ROWA-A class of protocols are more resilient to failures, including network partitioning, than the simple ROWA protocol.

Another class of eager replication protocols are those based on voting. The fundamental characteristics of voting were presented in the previous chapter when we discussed network partitioning in nonreplicated databases. The general ideas hold in the replicated case. Fundamentally, each read and write operation has to obtain a sufficient number of votes to be able to commit. These protocols can be pessimistic or optimistic. In what follows we discuss only pessimistic protocols. An optimistic version [Davidson, 1984] compensates transactions to recover if the commit decision cannot be confirmed at completion. This version is suitable wherever compensating transactions are acceptable (see Chapter 10).

The initial voting algorithm was proposed by Thomas [1979] and an early suggestion to use quorum-based voting for replica control is due to Gifford [1979]. Thomas's algorithm works on fully replicated databases and assigns an equal vote to each site. For any operation of a transaction to execute, it must collect affirmative votes from a majority of the sites. Gifford's algorithm, on the other hand, works with partially replicated databases (as well as with fully replicated ones) and assigns a vote to each copy of a replicated data item. Each operation then has to obtain a *read quorum* (V_r) or a *write quorum* (V_w) to read or write a data item, respectively. If a given data item has a total of V votes, the quorums have to obey the following rules:

1. $V_r + V_w > V$
2. $V_w > V/2$

As the reader may recall from the preceding chapter, the first rule ensures that a data item is not read and written by two transactions concurrently (avoiding the read-write conflict). The second rule, on the other hand, ensures that two write operations from two transactions cannot occur concurrently on the same data item (avoiding write-write conflict). Thus the two rules ensure that serializability and one-copy equivalence are maintained.

In the case of network partitioning, the quorum-based protocols work well since they basically determine which transactions are going to terminate based on the votes that they can obtain. The vote allocation and threshold rules given above ensure that two transactions that are initiated in two different partitions and access the same data cannot terminate at the same time.

The difficulty with this version of the protocol is that transactions are required to obtain a quorum even to read data. This significantly and unnecessarily slows down read access to the database. We describe below another quorum-based voting protocol [Abbadi et al., 1985] that overcomes this serious performance drawback.

The protocol makes certain assumptions about the underlying communication layer and the occurrence of failures. The assumption about failures is that they are "clean." This means two things:

1. Failures that change the network's topology are detected by all sites instantaneously.

2. Each site has a view of the network consisting of all the sites with which it can communicate.

Based on the presence of a communication network that can ensure these two conditions, the replica control protocol is a simple implementation of the ROWA-A principle. When the replica control protocol attempts to read or write a data item, it first checks if a majority of the sites are in the same partition as the site at which the protocol is running. If so, it implements the ROWA rule within that partition: it reads any copy of the data item and writes all copies that are in that partition.

Notice that the read or the write operation will execute in only one partition. Therefore, this is a pessimistic protocol that guarantees one-copy serializability, *but only within that partition*. When the partitioning is repaired, the database is recovered by propagating the results of the update to the other partitions.

A fundamental question with respect to implementation of this protocol is whether or not the failure assumptions are realistic. Unfortunately, they are not. Most network failures are not "clean." There is a time delay between the occurrence of a failure and its detection by a site. Because of this delay, it is possible for one site to think that it is in one partition when in fact subsequent failures have placed it in another partition. Furthermore, this delay may be different for various sites. Thus two sites that were in the same partition but are now in different partitions may proceed for a while under the assumption that they are still in the same partition. The violations of these two failure assumptions have significant negative consequences on the replica control protocol and its ability to maintain one-copy serializability.

The suggested solution is to build on top of the physical communication layer another layer of abstraction which hides the "unclean" failure characteristics of the physical communication layer and presents to the replica control protocol a communication service that has "clean" failure properties. This new layer of abstraction provides *virtual partitions* within which the replica control protocol operates. A virtual partition is a group of sites that have agreed on a common view of who is in that partition. Sites join and depart from virtual partitions under the control of this new communication layer, which ensures that the clean failure assumptions hold.

The advantage of this protocol is its simplicity. It does not incur any overhead to maintain a quorum for read accesses. Thus the reads can proceed as fast as they would in a nonpartitioned network. Furthermore, it is general enough so that the replica control protocol does not need to differentiate between site failures and network partitions.

Given alternative methods for achieving fault-tolerance in the case of replicated databases, a natural question is what the relative advantages of these methods are. There have been a number of studies that analyze these techniques, each with varying assumptions. A comprehensive study suggests that ROWA-A implementations achieve better scalability and availability than quorum techniques [Jiménez-Peris et al., 2003].

13.6 REPLICATION MEDIATOR SERVICE

The replication protocols we have so far covered are suitable for tightly integrated distributed database systems where we can insert the protocols into each component DBMS. In multidatabase systems, replication support has to be supported outside the DBMSs by mediators. In this section we discuss how to provide replication support at the mediator level by means of an example protocol due to Patiño-Martínez et al. [2000].

For this discussion, we will assume a fully replicated database for simplicity. The protocol called NODO (NOn-Disjoint conflict classes and Optimistic multicast) [Patiño-Martínez et al., 2000] follows the single master approach: it executes transactions at only one site and propagates only their write operations to the secondary copies at the slave sites. To reduce latency, this propagation employs an optimistic multicast technique that uses basic multicast.

The protocol exploits the knowledge of transactions' conflict classes to increase concurrency. A *conflict class* refers to a copy or a copy's horizontal fragment (i.e. subset of tuples) which is accessed by a transaction in update mode. Thus, two transactions that access the same conflict class have a high probability of conflict, thereby hurting parallelism, while two transactions that access different conflict classes can run in parallel. A transaction can access several conflict classes which must be statically known before execution (e.g. by analyzing the transaction code). Thus, conflict classes are further abstracted in conflict class groups. Each conflict class group has a single master node where all transactions on that conflict class group must be executed. Since the same conflict class can be in different conflict class groups at different nodes, distributed replication is possible. For instance, let node N be the master of $\{C_x, C_y\}$ and node N' the master of $\{C_y\}$, transactions T_1 on C_x, C_y and T_2 on C_y are executed at N and N', respectively.

We present a simplified form of the protocol. Each transaction is assumed to update a single conflict class group, at a single master node, i.e. the transaction is not distributed. This transaction is said *local* to the node (its master node). Each node manages a number of queues for its incoming transactions, one per conflict class. When a transaction arrives at a master node, it is processed by all nodes (including the master node) in three steps: (1) an optimistic total-ordered multicast to execute the transaction at the master node (when the transaction is optimistically delivered), (2) and to commit the transaction (when the transactions is definitively delivered), (3) a basic ordered multicast to apply the transaction's write set at non-master nodes. Optimistic multicast yields an arbitrary order (like basic multicast discussed earlier) with the additional guarantee that a message optimistically delivered does not precede a message delivered in total order

Example 13.6

Let node N, respectively N', be the master of the conflict class group $\{C_x, C_y\}$, respectively$\{C_y\}$. Let transaction T_1 be on $\{C_x, C_y\}$, T_2 on $\{C_y\}$ and T_3 on $\{C_x\}$. at N and N'. Let transaction T_i the i-th in the total order. Thus, T_1 and T_3 are local to N while T_2 is local to N'. Consider the following state of the queues C_x and C_y at N and N' after the transactions have been opt-delivered.

$$N: \quad C_x = [T_1, T_3]; C_y = [T_1, T_2]$$
$$N': \quad C_x = [T_3, T_1]; C_y = [T_2, T_1]$$

Let us consider transaction execution at N. T_1 is ahead of the queues C_x and C_y and thus can be executed. When N to-delivers T_1, it can commit T_1 (since total order is respected) and remove from its queues. Then, it multicasts the commit message for T_1. Since there is no conflict between T_2 and T_3, T_3 can be similarly executed and committed before or after T_2. Let us assume that T_2 is executed and committed before T_3. Let us now consider transaction execution at N'. T_2 is ahead of the queue C_y and thus can be executed. When N' receives the to-delivered message for T_1, it detects that T_2's execution has violated total order. Thus, it aborts T_2's and reoders the queues so that T_2 is at the end of queue C_y and T_1 is ahead of both queues C_x and C_y. When N' receives the commit message for T_1, now ahead of the queues, it can execute and commit T_1 and remove it from the queues. Then, T_2 followed by T_3 can be executed and committed. The final ordering is T_1, T_2, T_3 at both nodes.

Interestingly, there are many cases where in spite of an ordering mismatch between optimistic and definitive delivery, it is possible to commit transactions consistently by using the optimistic rather than total ordering, thus minimizing the number of aborts due to optimism failures [Patiño-Martínez et al., 2005].

The implementation of the NODO protocol combines concurrency control with group communication primitives and what has been traditionally done inside the DBMS. This solution can be implemented outside a DBMS without a negligible overhead, and thus supports DBMS autonomy Jiménez-Peris et al. [2002]. Similar eager replication protocols have been proposed to support *partial replication*, where copies can be stored at subsets of nodes [Sousa et al., 2001]. Unlike full replication, partial replication increases access locality and reduces the number of messages for propagating updates to replicas.

13.7 CONCLUSION

In this chapter we discussed different approaches to data replication and presented protocols that are appropriate under different circumstances. Each of the alternative protocols we have discussed have their advantages and disadvantages. Eager centralized protocols are simple to implement, they do not require update coordination across sites, and they are guaranteed to lead to one-copy serializable histories. However, they put a significant load on the master sites, potentially causing them to be bottleneck. Consequently, they are harder to scale, in particular in the single master site architecture – primary copy versions have better scalability properties since the master responsibilities are somewhat distributed. These protocols result in long response times (the longest among the four alternatives), since the access to any data has to wait until the commit of any transaction that is currently updating it (using 2PC, which is expensive). Furthermore, the local copies are used sparingly, only for read operations. Thus, if the workload is update-intensive, eager centralized protocols are likely to suffer from bad performance.

Eager distributed protocols also guarantee one-copy serializability and provide an elegant symmetric solution where each site performs the same function. However, unless there is communication system support for efficient multicasting, they result in very high number of messages that increase network load and result in high transaction response times. This also constrains their scalability. Furthermore, naive implementations of these protocols will cause significant number of deadlocks since update operations are executed at multiple sites concurrently.

Lazy centralized protocols have very short response times since transactions execute and commit at the master, and do not need to wait for completion at the slave sites. There is also no need to coordinate across sites during the execution of an update transaction, thus reducing the number of messages. On the other hand, mutual consistency (i.e., freshness of data at all copies) is not guaranteed as local copies can be out of date. This means that it is not possible to do a local read and be assured that the most up-to-date copy is read.

Finally, lazy multi-master protocols have the shortest response times and the highest availability. This is because each transaction is executed locally, with no distributed coordination. Only after they commit are the other replicas updated through refresh transactions. However, this is also the shortcoming of these protocols – different replicas can be updated by different transactions, requiring elaborate reconciliation protocols and resulting in lost updates.

Replication has been studied extensively within the distributed computing community as well as the database community. Although there are considerable similarities in the problem definition in the two environments, there are also important differences. Perhaps the two more important differences are the following. Data replication focuses on data, while replication of computation is equally important in distributed computing. In particular, concerns about data replication in mobile environments that involve disconnected operation have received considerable attention. Secondly, database and transaction consistency is of paramount importance in data replication; in distributed computing, consistency concerns are not as high on the list of priorities. Consequently, considerably weaker consistency criteria have been defined.

Replication has been studied within the context of parallel database systems, in particular within parallel database clusters. We discuss these separately in Chapter 14.

13.8 BIBLIOGRAPHIC NOTES

Replication and replica control protocols have been the subject of significant investigation since early days of distributed database research. This work is summarized very well in [Helal et al., 1997]. Replica control protocols that deal with network partitioning are surveyed in [Davidson et al., 1985].

A landmark paper that defined a framework for various replication algorithms and argued that eager replication is problematic (thus opening up a torrent of activity on lazy techniques) is [Gray et al., 1996]. The characterization that we use in this chapter is based on this framework. A more detailed characterization is given in Wiesmann et al. [2000]. A recent survey on optimistic (or lazy) replication techniques is [Saito and Shapiro, 2005].

Freshness, in particular for lazy techniques, have been a topic of some study. Alternative techniques to ensure "better" freshness are discussed in [Pacitti et al., 1998; Pacitti and Simon, 2000; Röhm et al., 2002b; Pape et al., 2004a; Akal et al., 2005].

There are many different versions of quorum-based protocols. Some of these are discussed in [Triantafillou and Taylor, 1995; Paris, 1986; Tanenbaum and van Renesse, 1988]. Besides the algorithms we have described here, some notable others are given in [Davidson, 1984; Eager and Sevcik, 1983; Herlihy, 1987; Minoura and Wiederhold, 1982; Skeen and Wright, 1984; Wright, 1983]. These algorithms are generally called *static* since the vote assignments and read/write quorums are fixed a priori. An analysis of one such protocol (such analyses are rare) is given in [Kumar and Segev, 1993]. Examples of *dynamic replication protocols* are in [Jajodia and Mutchler, 1987; Barbara et al., 1986, 1989] among others. It is also possible to change the way data are replicated. Such protocols are called *adaptive* and one example is described in [Wolfson, 1987].

An interesting replication algorithm based on economic models is described in [Sidell et al., 1996].

13.9 EXERCISES

13.1 Consider data items x and y replicated across the sites as follows:

Site 1	Site 2	Site 3	Site 4
x	x		x
	y	y	y

(a) Assign votes to each site and give the read and write quorum.

(b) Determine the possible ways that the network can partition and for each specify in which group of sites a transaction that updates (reads and writes) x can be terminated and what the termination condition would be.

(c) Repeat (**b**) for y.

****13.2** In the NODO protocol, we have seen that each conflict class group has a master. However, this is not inherent to the protocol. Design a multi-master variation of NODO in which a transaction might be executed by any replica. What condition should be enforced to guarantee that each updated transaction is processed only by one replica?

****13.3** In the NODO protocol, if the DBMS could provide additional introspection functionality, it would be possible to execute in certain circumstances transactions of the same conflict class in parallel. Determine which functionality would be needed from the DBMS. Also characterize formally under which circumstances concurrent execution of transactions in the same conflict class could be allowed to be executed in parallel whilst respecting 1-copy consistency. Extend the NODO protocol with this enhancement.