8
REPLICATED DATA

8.1 INTRODUCTION

A replicated database is a distributed database in which multiple copies of some data items are stored at multiple sites. The main reason for using replicated data is to increase DBS availability. By storing critical data at multiple sites, the DBS can operate even though some sites have failed. Another goal is improved performance. Since there are many copies of each data item, a transaction is more likely to find the data it needs close by, as compared to a single copy database. This benefit is mitigated by the need to update all copies of each data item. Thus, Reads may run faster at the expense of slower Writes.

Our goal is to design a DBS that hides all aspects of data replication from users' transactions. That is, transactions issue Reads and Writes on data items, and the DBS is responsible for translating those operations into Reads and Writes on one or more copies of those data items. Before looking at the architecture of a DBS that performs these functions, let's first determine what it means for such a system to behave correctly.

Correctness

We assume that a DBS managing a replicated database should behave like a DBS managing a one-copy (i.e., nonreplicated) database insofar as users can tell. In a one-copy database, users expect the interleaved execution of their

transactions to be equivalent to a serial execution of those transactions. Since replicated data should be transparent to them, they would like the interleaved execution of their transactions on a *replicated* database to be equivalent to a serial execution of those transactions on a *one-copy* database. Such executions are called *one-copy serializable* (or *1SR*). This is the goal of concurrency control for replicated data.

This concept of one-copy serializability is essentially the same as the one we used for multiversion data in Chapter 5. In both cases we are giving the user a one-copy view of a database that may have multiple copies (replicated copies or multiple versions) of each data item. The only difference is that here we are abstracting replicated copies, rather than multiple versions, from the users' view.

The Write-All Approach

In an ideal world where sites never fail, a DBS can easily manage replicated data. It translates each Read(x) into Read(x_A), where x_A is any copy of data item x (x_A denotes the copy of x at site A). It translates each Write(x) into {Write(x_{A_1}), . . ., Write(x_{A_m})}, where $\{x_{A_1}, \ldots, x_{A_m}\}$ are all copies of x. And it uses any serializable concurrency control algorithm to synchronize access to copies. We call this the *write-all approach* to replicated data.

To see why the write-all approach works, consider any execution produced by the DBS. Since the DBS is using a serializable concurrency control algorithm, this execution is equivalent to some serial execution. In that serial execution, each transaction that writes into a data item x writes into all copies of x. From the viewpoint of the next transaction in the serial execution, all copies of x were written simultaneously. So, no matter which copy of x the next transaction reads, it reads the same value, namely, the one written by the last transaction that wrote all copies of x. Thus, the execution behaves as though it were operating on a single copy database.

Unfortunately, the world is less than ideal — sites can fail and recover. This is a problem for the write-all approach, because it requires that the DBS process each Write(x) by writing into *all* copies of x, even if some have failed. Since there will be times when some copies of x are down, the DBS will not always be able to write into all copies of x at the time it receives a Write(x) operation. If the DBS were to adhere to the write-all approach in this situation, it would have to delay processing Write(x) until it could write into all copies of x.

Such a delay is obviously bad for update transactions. If any copy of x fails, then no transaction that writes into x can execute to completion. The more copies of x that exist, the higher the probability that one of them is down. In this case, more replication of data actually makes the system less

available to update transactions! For this reason, the write-all approach is unsatisfactory.

The Write-All-Available Approach

Suppose we adopt a more flexible approach. We still require the DBS to produce a serializable execution, but no longer require a transaction to write into all copies of each data item x in its writeset. It *should* write into all of the copies that it can, but it may ignore any copies that are down or not yet created. This is the *write-all-available approach*. It solves the availability problem, but may lead to problems of correctness.

Using the write-all-available approach, there will be times when some copies of x do not reflect the most up-to-date value of x. A transaction that reads an out-of-date copy of x can create an incorrect, i.e., non-1SR, execution. The following execution, H_1, shows how this can happen:

$$H_1 = w_0[x_A]\ w_0[x_B]\ w_0[y_C]\ c_0\ r_1[y_C]\ w_1[x_A]\ c_1\ r_2[x_B]\ w_2[y_C]\ c_2.$$

Notice that T_2 read copy x_B of x from T_0, even though T_1 was the last transaction before it that wrote into x. That is, T_2 read an out-of-date copy of x.

In a serial execution on a one-copy database, if a transaction reads a data item x, then it reads x from the last transaction before it that wrote into x. But this is not what happened in H_1. T_2 read x_B from T_0, which is *not* the last transaction before it that wrote into x. Thus H_1 is not equivalent to the serial execution $T_0\ T_1\ T_2$ on a one-copy database. We could still regard H_1 as correct if it were equivalent to another serial execution on a one-copy database. However, since $w_0[y_C] < r_1[y_C] < w_2[y_C]$, there are no other serial executions equivalent to H_1. Therefore, H_1 is not equivalent to any serial execution on a one-copy database. That is, H_1 is not 1SR.

T_1 seems to be the culprit here, because it did not write into *all* copies of x. Unfortunately, it may have had no choice. For example, suppose site B failed after T_0 but before T_1, and recovered after T_1 but before T_2, as in H_1':

$$H_1' = w_0[x_A]\ w_0[x_B]\ w_0[y_C]\ c_0\ B\text{-fails}\ r_1[y_C]\ w_1[x_A]\ c_1\ B\text{-recovers}\ r_2[x_B]\ w_2[y_C]\ c_2.$$

Rather than waiting for B to recover so it could write x_B, T_1 wrote into the one copy that it *could* write, namely, x_A. After B recovered, T_2 unwittingly read x_B, an out-of-date copy of x, and therefore produced an incorrect result. This particular problem could be easily solved by preventing transactions from reading copies from sites that have failed and recovered until these copies are brought up-to-date. Unfortunately, this isn't enough, as we'll see later (cf. Section 8.4.).

There are several algorithms, including some variations of the write-all-available approach, that correctly handle failures and recoveries and thereby avoid incorrect executions such as H_1. These algorithms are the main subject

of this chapter. But before we delve deeply into this subject, let's first define a system architecture for DBSs that manage replicated data.

8.2 SYSTEM ARCHITECTURE

We will assume that the DBS is distributed. As usual, each site has a data manager (DM) and transaction manager (TM) that manage data and transactions at the site.

The Data Manager

The DM is a centralized DBS that processes Reads and Writes on local copies of data items. It has an associated local scheduler for concurrency control, based on one of the standard techniques (2PL, TO, or SGT). In addition, there may be some interaction between local schedulers, for example, to detect distributed deadlocks or SG cycles.

As in Chapter 7, we assume that the DM and scheduler at a site are able to commit a transaction's Writes that were executed at that site. By committing a transaction T, the scheduler guarantees that the recoverability condition holds for all of T's Reads at that site, and the DM guarantees that all of T's Writes at that site are in stable storage (i.e., it satisfies the Redo Rule).

The scheduler at a site is only sensitive to conflicts between operations on the same *copy* of a data item. For example, if the scheduler at site A receives an operation $r_i[x_A]$, it will synchronize $r_i[x_A]$ relative to Writes it has received on x_A. However, since it doesn't receive Writes on other copies of x, it cannot synchronize $r_i[x_A]$ relative to Writes on those other copies. The scheduler is really treating operations on copies as if they were operations on independent data items. In this sense, it is entirely oblivious to data replication.

The Transaction Manager

The TM is the interface between user transactions and the DBS. It translates users' Reads and Writes on data items into Reads and Writes on copies of those data items. It sends those Reads and Writes on copies to the appropriate sites, where they are processed by the local schedulers and DMs.

The TM also uses an atomic commitment protocol (ACP), so that it can consistently terminate a transaction that accessed data at more than one site. We assume that the DM and scheduler at each site are designed to participate in this ACP for any transaction that is active at that site.

To perform its functions, the TM must determine which sites have copies of which data items. It uses *directories* for this purpose.[1] There may be just one

[1]Note that the term *directory* has a different meaning here than in Chapter 6. In this chapter, a directory maps each data item x to the sites that have copies of x. In Chapter 6, it mapped each data item to its stable storage location.

directory that tells where all copies of all data items are stored. The directory can be stored at all sites or only at some of them. Alternatively, each directory may only give the location of copies of some of the data items. In this case, each directory is normally only stored at those sites that frequently access the information that it contains. To find the remaining directories, the TM needs a *master directory* that tells where copies of each directory are located.

To process a transaction, a TM must access the directories that tell it where to find the copies of data items that the transaction needs. If communication is expensive, then the directories should be designed so that each TM will usually be able to find a copy of those directories at its own site. Otherwise, the TM will have to send messages to other sites to find the directories it needs.

Failure Assumptions

We say that a copy x_A of a data item or directory at site A is *available to site B* if A correctly executes each Read and Write on x_A issued by B and B receives A's acknowledgment of that execution. Thus copy x_A may be unavailable to B for one of three reasons:

1. A does not receive Reads and Writes on x_A issued by B. In this case, by definition, a communications failure has occurred (see Section 7.2).
2. The communication network correctly delivers to A Reads and Writes on x_A issued by B, but A is unable to execute them, either because A is down or A has suffered a failure of the storage medium that contains x_A.
3. A receives and executes each Read and Write issued by B, but B does not receive A's acknowledgment of such executions (due to a communications failure).

We say a copy x_A is *available* (or *unavailable*) if it is available (or not available) to every site other than A.

As in Chapter 7, we assume that sites are fail-stop, and that site failures are detected by timeout. Thus, in the absence of communications failures, each site can determine whether any other site is failed simply by sending a message and waiting for a reply. That is, site failures are *detectable*.

If communications failures may occur, then a site A that is not responding to messages may still be functioning. This creates nasty problems for managing replicated data, because A may try to read or write its copies without being able to synchronize against Reads or Writes on copies of the same data item at other sites.

Distributing Writes

When a transaction issues Write(x), the DBS is responsible for eventually updating a set of copies of x (the exact set depends on the algorithm used for

managing replicated data). It can distribute these Writes *immediately*, at the moment it receives Write(x) from the transaction. Or, it can *defer* the Writes on replicated copies until the transaction terminates.

With deferred writing, the DBS uses a nonreplicated view of the database while the transaction is executing. That is, for each data item that the transaction reads or writes, the DBS accesses one and only one copy of that data item. (Different transactions may use different copies.) The DBS delays the distribution of Writes to other copies until the transaction has terminated and is ready to commit. The DBS must therefore maintain an intentions list of deferred updates.[2] After the transaction terminates, it sends the appropriate portion of the intentions list to each site that contains replicated copies of the transaction's writeset, but that has not yet received those Writes. It can piggyback this message with the VOTE-REQ message of the first phase of the ACP.

Except for the copies it uses while executing the transaction, a DBS that uses deferred writing puts all replicated Writes destined for the same site in a single message. This tends to minimize the number of messages required to execute a transaction. By contrast, using immediate writing, the DBS sends Writes to replicated copies while the transaction executes. Although some piggybacking may be possible, it essentially uses one message for each Write. Thus, immediate writing tends to use more messages than deferred writing.

Another advantage of deferred writing is that Aborts often cost less than with immediate writing. In a DBS that uses immediate writing, when a transaction T_i aborts, the DBS is likely to have already distributed many of T_i's Writes to replicated copies. Not only are these Writes wasted, but they also must be undone. With deferred writing, the DBS delays the distribution of those Writes until after T_i has terminated. If T_i aborts before it terminates, then the abortion is less costly than with immediate writing.

A disadvantage of deferred writing is that it may delay the commitment of a transaction more than immediate writing. This is because the first phase of the ACP at each receiving site must process a potentially large number of Writes before it can respond to the VOTE-REQ message. With immediate writing, receiving sites can execute many of a transaction's Writes while the transaction is still executing, thereby avoiding the delay of executing them at commit time.

A second disadvantage of deferred writing is that it tends to delay the detection of conflicts between operations. For example, suppose transactions T_1 and T_2 execute concurrently and both write into x. Furthermore, suppose the DBS uses copy x_A while executing T_1 and uses x_B while executing T_2. Until the DBS distributes T_1's replicated Write on x_B or T_2's replicated Write on x_A, no scheduler will detect the conflicting Writes between T_1 and T_2. With deferred writing, this happens at the end of T_1's and T_2's execution. This may be less desirable than immediate writing, since it may cause a scheduler to

[2]Cf. the no-undo/redo centralized recovery algorithm in Section 6.6.

reject a Write later in a transaction's execution. The DBS ends up aborting the transaction after having paid for most of the transaction's execution. (This is similar to a disadvantage of 2PL certifiers described in Section 4.4.)

This disadvantage of deferred writing can be mitigated by requiring the DBS to use the same copy of each data item, called the *primary copy*, to execute every transaction. For example, the DBS would use the same (primary) copy of x, x_A, to execute both T_1 and T_2. The scheduler for x_A detects the conflict between T_1's and T_2's writes, thereby detecting it earlier than if T_1 and T_2 used different copies of x. In this case, deferred writing and immediate writing detect the conflict at about the same point in a transaction's execution.

8.3 SERIALIZABILITY THEORY FOR REPLICATED DATA

We will extend basic serializability theory by using two types of histories: *replicated data* (*RD*) histories and *one-copy* (*1C*) histories. RD histories represent the DBS's view of executions of operations on a replicated database. 1C histories represent the interpretation of RD histories in the users' single copy view of the database. (1C histories are quite similar to the 1V histories we used in Chapter 5.) As usual, we will characterize a concurrency control algorithm by the RD histories it produces. To prove an algorithm correct, we prove that its RD histories are equivalent to serial 1C histories, which are the histories that the user regards as correct.

The formal development of serializability theory for replicated databases is very similar to that for multiversion databases. The notations and the formal notions of correctness are analogous. You may find it helpful to think about these similarities while you're reading this section.

Replicated Data Histories

Let $T = \{T_0, \ldots, T_n\}$ be a set of transactions. To process operations from T, a DBS translates T's operations on data items into operations on the replicated copies of those data items. We formalize this translation by a function h that maps each $r_i[x]$ into $r_i[x_A]$, where x_A is a copy of x; each $w_i[x]$ into $w_i[x_{A_1}], \ldots, w_i[x_{A_m}]$ for some copies $x_{A_1}, \ldots, x_{A_m}$ of x ($m > 0$); each c_i into c_i and each a_i into a_i.

A *complete replicated data* (*RD*) history H over $T = \{T_0, \ldots, T_n\}$ is a partial order with ordering relation $<$ where

1. $H = h\,(\cup_{i=0}^{n} T_i)$ for some translation function h;
2. for each T_i and all operations p_i, q_i in T_i, if $p_i <_i q_i$, then every operation in $h(p_i)$ is related by $<$ to every operation in $h(q_i)$;
3. for every $r_j[x_A]$, there is at least one $w_i[x_A] < r_j[x_A]$;

4. all pairs of conflicting operations are related by $<$, where two operations *conflict* if they operate on the same *copy* and at least one of them is a Write; and
5. if $w_i[x] <_i r_i[x]$ and $h(r_i[x]) = r_i[x_A]$ then $w_i[x_A] \in h(w_i[x])$.

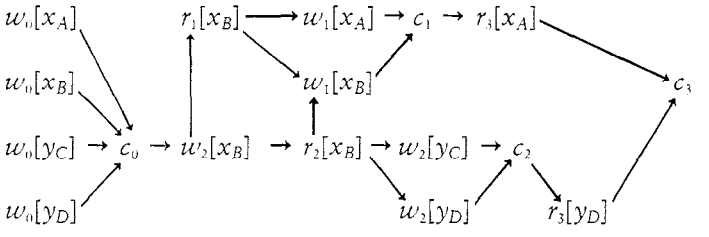
Condition (1) states that the DBS translates each operation submitted by a transaction into appropriate operations on copies. Condition (2) states that the RD history preserves all orderings stipulated by transactions. Condition (3) states that a transaction may not read a copy unless it has been previously initialized. Condition (4) states that the history records the execution order of noncommutative operations. Condition (5) states that if a transaction T_i writes into a data item x before it reads x, then it must write into the same copy x_A of x that it subsequently reads. (It may write into other copies of x as well.) By Condition (2), this implies $w_i[x_A] < r_i[x_A]$. We will revisit this issue in a moment.

Given transactions $\{T_0, T_1, T_2, T_3\}$:

$$T_0 = \begin{matrix} w_0[x] \searrow \\ w_0[y] \rightarrow c_0 \end{matrix} \qquad T_2 = w_2[x] \rightarrow r_2[x] \rightarrow w_2[y] \rightarrow c_2$$

$$T_1 = r_1[x] \rightarrow w_1[x] \rightarrow c_1 \qquad T_3 = \begin{matrix} r_3[x] \searrow \\ r_3[y] \rightarrow c_3 \end{matrix}$$

the following history, H_2, is an RD history over $\{T_0, \ldots, T_3\}$:



Let H be an RD history over T. Transaction T_j *reads-x-from* T_i in H if for some copy x_A T_j reads-x_A-from T_i, that is, if $w_i[x_A] < r_j[x_A]$ and no $w_k[x_A]$ $(k \neq i)$ falls between these operations. Since reads-from relationships on copies are unique by condition (4) of RD histories and since a transaction reads at most one copy of any given data item,[3] reads-from relationships on data items are unique as well.

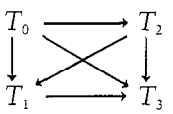
In this chapter, *we will only consider histories that are complete and that only contain committed transactions* (i.e., histories that are committed projections of themselves). These restrictions are justified by requiring that we only

[3]This follows from the fact that $h(r_i[x]) = r_i[x_A]$ (i.e., $h(r_i[x])$ does not equal a set of Reads) and from the assumption that a transaction reads a data item at most once. As we mentioned in Chapter 2, the latter assumption is mainly a convenience that can be dispensed with at some cost in notational complexity.

use schedulers that produce recoverable executions. Recoverability for RD histories is defined as for 1C histories, but with respect to copies. That is, an RD history H is recoverable if whenever T_i reads (any copy) from T_j in H and $c_i \in H$, then $c_j < c_i$. We assume that all RD histories are recoverable.

The serialization graph for an RD history is defined as for a 1C history. That is, the nodes correspond to committed transactions in the history and there is an edge $T_i \rightarrow T_j$ if there are conflicting operations p_i in T_i and q_j in T_j such that $p_i < q_j$. Thus, the serialization graph for H_2 is

$\mathrm{SG}(H_2) =$



If SG(H) is acyclic, then conditions (2) and (5) in the definition of RD history ensure that H *preserves reflexive reads-from relationships*. More precisely, we have:

Lemma 8.1: Let H be an RD history involving transaction T_i. If SG(H) is acyclic and for some x $w_i[x] <_i r_i[x]$, then T_i reads-x-from T_i in H.

Proof: From conditions (2) and (5) on RD histories, $w_i[x] <_i r_i[x]$ implies that for some copy x_A of x, $w_i[x_A] < r_i[x_A]$. Suppose, by way of contradiction, T_i didn't read x from T_i in H. Then there must exist some $w_k[x_A]$ $(k \neq i)$ in H such that $w_i[x_A] < w_k[x_A] < r_i[x_A]$. But then SG($H$) contains edges $T_i \rightarrow T_k$ and $T_k \rightarrow T_i$ which contradicts the assumed acyclicity of SG(H). □

We would like to define an RD history H to be 1SR if it is equivalent to a serial 1C history H_{1C}. To determine if H is equivalent to H_{1C}, it would be unsatisfactory to use the notion of conflict equivalence, simply because H and H_{1C} have different operations.[4] However, reads-from relationships and final writes do behave the same way in both types of histories. Therefore, view equivalence provides a natural way to determine the equivalence of an RD history and 1C history (see Section 2.6 for a discussion of view equivalence in 1C histories).

Given RD history H, define $w_i[x_A]$ to be a *final write for* x_A in H if $a_i \notin H$ and for all $w_j[x_A]$ in H $(j \neq i)$, either $a_j \in H$ or $w_j[x_A] < w_i[x_A]$.

Two RD histories over T are *equivalent* (denoted $\equiv$) if they are view equivalent, that is, if they have the same reads-from relationships and final writes.

---

[4]That is, if we define H and H_{1C} to be equivalent if conflicting operations appear in the same order in both histories, then we must deal with the problem that some operations that conflict in H_{1C} may not have corresponding operations that conflict in H. For example, consider $H = w_2[x_B]\ w_1[x_A]$ and $H_{1C} = w_2[x]\ w_1[x]$.

An RD history H over T is equivalent to a 1C history H_{1C} over T if

1. H and H_{1C} have the same reads-from relationships *on data items* (i.e., T_j reads-x-from T_i in H iff the same holds in H_{1C}), and
2. for each final write $w_i[x]$ in H_{1C}, $w_i[x_A]$ is a final write in H for some copy x_A of x.[5]

An RD history is *one-copy serializable* (*1SR*) if it is equivalent to a serial 1C history. For example, H_2 is 1SR since it is equivalent to the following 1C history:

$$H_3 = w_0[x]\ w_0[y]\ c_0\ w_2[x]\ r_2[x]\ w_2[y]\ c_2\ r_1[x]\ w_1[x]\ c_1\ r_3[x]\ r_3[y]\ c_3.$$

History H_1 in Section 8.1, reproduced below, is not 1SR:

$$H_1 = w_0[x_A]\ w_0[x_B]\ w_0[y_C]\ c_0\ r_1[y_C]\ w_1[x_A]\ c_1\ r_2[x_B]\ w_2[y_C]\ c_2.$$

Notice that H_1 is a serial RD history. Thus, not every serial RD history is 1SR.

Ignoring Final Writes

All of the concurrency control algorithms for replicated databases that we will study use conflict-based schedulers. That is, they use schedulers that produce histories that have acyclic serialization graphs. We can use this fact to avoid dealing with final writes in proving the equivalence of RD and 1C histories, as described in the following lemma.

Lemma 8.2: Let H be an RD history over T, where SG(H) is acyclic. Let H_{1C} be a serial 1C history over T such that the order of transactions in H_{1C} is consistent with SG(H). (That is, if $T_i \rightarrow T_j$ is in SG(H), then T_i precedes T_j in H_{1C}.) If $w_i[x]$ is a final write for x in H_{1C}, then every Write, $w_i[x_A]$, by T_i into some copy x_A of x is a final write for x_A in H.

Proof: Suppose $w_i[x]$ is a final write for x in H_{1C}. Let $w_i[x_A]$ be any Write into x by T_i in H. If $w_i[x_A]$ is not a final write, then there is some $w_j[x_A]$ $(j \neq i)$ such that $a_j \notin H$ and $w_i[x_A] < w_j[x_A]$. Thus, $T_i \rightarrow T_j$ is in SG(H), so T_i precedes T_j in H_{1C}. But that implies $a_j \notin H_{1C}$ and $w_i[x] < w_j[x]$ in H_{1C}, contradicting the choice of $w_i[x]$ as a final write. □

To prove that an RD history H is equivalent to a serial 1C history H_{1C}, we would ordinarily have to prove that H and H_{1C} have the same reads-from relationships and final writes. However, if transactions in H_{1C} are in an order

[5]Recall from Section 2.6 that $w_i[x]$ is a *final write for* x in H_{1C} if $a_i \notin H_{1C}$ and for all $w_j[x] \in H_{1C}$ $(j \neq i)$, either $a_j \in H$ or $w_j[x] < w_i[x]$. Notice that final writes are defined for Writes on data items in 1C histories and for Writes on *copies* of data items in RD histories. Hence the need for the special wording of condition (2) in the definition of equivalence of H and H_{1C}, instead of simply saying they have the same reads-from relationships and final writes.

consistent with SG(H) (which, since H_{1C} is serial, implies that SG(H) is acyclic), then by Lemma 8.2 we only have to prove they have the same reads-from relationships. That is, we have the following theorem.

Theorem 8.3: Let H be an RD history. If H has the same reads-from relationships as a serial 1C history H_{1C}, where the order of transactions in H_{1C} is consistent with SG(H), then H is 1SR. □

8.4 A GRAPH CHARACTERIZATION OF 1SR HISTORIES

To prove the correctness of a concurrency control algorithm for replicated data, we must prove that all of the histories it produces are 1SR. Following our usual approach, we would like to do this with the help of a serialization graph structure. Unfortunately, standard serialization graphs for RD histories are too weak for this purpose, as illustrated by the following example.

Consider a database with data items x and y and copies x_A, x_B, y_C, and y_D. Suppose we have the following three transactions:

$$T_0 = \begin{matrix} w_0[x] \\ w_0[y] \end{matrix} \rightarrow c_0 \qquad T_1 = r_1[x] \rightarrow w_1[y] \rightarrow c_1$$

$$T_2 = r_2[y] \rightarrow w_2[x] \rightarrow c_2$$

If the DBS uses 2PL and ignores failed copies, the following history can occur.

$$H_4 = \begin{matrix} w_0[x_A] \\ w_0[x_B] \\ w_0[y_C] \\ w_0[y_D] \end{matrix} \rightarrow c_0 \rightarrow \begin{matrix} r_1[x_A] \rightarrow y_D\sqcap \rightarrow w_1[y_C] \rightarrow c_1 \\ r_2[y_D] \rightarrow x_A\sqcap \rightarrow w_2[x_B] \rightarrow c_2 \end{matrix}$$

where $x_A\sqcap$ denotes the failure of copy x_A. We'll find it more convenient to speak of copies, rather than sites, failing. Transactions T_1 and T_2 begin by reading x_A and y_D. After they complete their Reads, the copies that they read fail. Then they perform their Writes. Since y_D is down, y_C is the only available copy of y. So the DBS translates $w_1[y]$ into $w_1[y_C]$. Similarly, since x_A is down, it translates $w_2[x]$ into $w_2[x_B]$. If the DBS uses 2PL by setting locks on copies that it accesses, it has no trouble locking each copy that T_1 and T_2 access, because no two operations of these transactions access the same copy of any data item.

H_4 is not 1SR. A serial execution of T_0, T_1, and T_2 on a single copy database would have either T_1 reading the value of x written by T_2, or T_2 reading the value of y written by T_1. However, in H_4 neither transaction reads the data written by the other. So, H_4 is not equivalent to a serial history over $\{T_0, T_1, T_2\}$ on a one-copy database. But despite the fact that H_4 is incorrect, SG(H_4) is acyclic:

$$SG(H_4) = \quad T_0 \longrightarrow T_1 \qquad T_2.$$

This example illustrates the rather surprising fact that the write-all-available approach may lead to an incorrect execution, even if only failures, but no recoveries occur (i.e., failed copies never recover).

One explanation of the problem in H_4 relates to the fundamental technique that all concurrency control algorithms use to obtain correct executions, namely, controlling the order of conflicting operations on shared data. In H_4, even though T_1 and T_2 have conflicting accesses to data items x and y, they don't have conflicting accesses to *copies* of x and y. The copy of x and y that each of them read failed before the other transaction's conflicting Write into that copy could be issued. The copy that each of them wrote is a copy that the other transaction didn't read. Thus, the logical conflicts were never manifested as physical conflicts on copies, which is the only place that the DBS can control them.

In a sense, the problem is finding a way to ensure that any two transactions that have conflicting accesses to the same data item also have conflicting accesses to some copy of that data item. That way, the DBS will be able to synchronize the transactions. If the DBS can do this in all cases, then by attaining ordinary serializability it has also attained one-copy serializability.

*Replicated Data Serialization Graphs

To determine if an RD history is 1SR, we will use a modified SG. This graph models the observation that two transactions that have conflicting accesses to the same data item must be synchronized, even if they don't access the same copy of that data item. To define this graph, we need a little terminology. We say that node n_i *precedes* node n_j, denoted $n_i \lll n_j$, in a directed graph if there is a path from n_i to n_j.

Given an RD history H, a *replicated data serialization graph* (*RDSG*) for H is SG(H) with enough edges added (possibly none) such that the following two conditions hold: For all data items x,

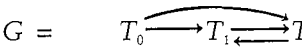
1. if T_i and T_k write x, then either $T_i \lll T_k$ or $T_k \lll T_i$), and
2. if T_j reads-x-from T_i, T_k writes some copy of x ($k \neq i$, $k \neq j$)), and $T_i \lll T_k$, then $T_j \lll T_k$.

If a graph satisfies condition (1), we say it *induces a write order* for H. If it satisfies condition (2), we say it *induces a read order* for H. Thus RDSG(H) is an extension of SG(H) that induces a read order and a write order for H. Note that an RDSG for H isn't uniquely determined by H.

We explained that failures can lead to incorrect behavior because transactions that have conflicting accesses to the same data item may not have conflicting accesses to copies of that data item. An SG doesn't have enough edges (due to conflicting accesses to copies) to force an order on every pair of transactions with conflicting accesses to the same data item. An RDSG for H adds enough edges to SG(H) to make this so. The write order ensures that the

RDSG orders every pair of transactions that write into the same data item, even if they don't write into the same copy. If the RDSG induces a write order, then the read order ensures that the RDSG orders every pair of transactions (T_j and T_k) that (respectively) read and write the same data item.

SG(H_4) is not an RDSG(H_4). It induces a write order for H_4, but not a read order. We can make it into an RDSG(H_4) by adding edges between T_1 and T_2.

$G =$ 

Since T_1 reads-x-from T_0, T_2 writes x, and $T_0 \rightarrow T_2$, we added $T_1 \rightarrow T_2$. Since T_2 reads-y-from T_0, T_1 writes y, and $T_0 \rightarrow T_1$, we added $T_2 \rightarrow T_1$. Since these are the only two reads-from relationships in H_4, these two edges are enough to ensure that G induces a read order for H. Notice that G has a cycle. Since every RDSG for H_4 must contain the edges $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$, every such RDSG has a cycle.

The following theorem is an important tool for analyzing the correctness of concurrency control algorithms for replicated data.

Theorem 8.4: Let H be an RD history. If H has an acyclic RDSG, then H is 1SR.

Proof: Let $H_s = T_{i_1} T_{i_2} \ldots T_{i_n}$ be a serial 1C history where $T_{i_1}, T_{i_2}, \ldots, T_{i_n}$ is a topological sort of RDSG(H). Since RDSG(H) contains SG(H), by Theorem 8.3, we can prove that H is 1SR just by proving that H and H_s have the same reads-from relationships.

First, assume T_j reads-x-from T_i in H. Suppose, by way of contradiction, that T_j reads-x-from T_k in H_s, for some $k \neq i$. If $k = j$, Lemma 8.1 implies that T_j reads-x-from T_k in H, a contradiction (note that since H has an acyclic RDSG, SG(H) is surely acyclic and therefore Lemma 8.1 applies). So, assume $k \neq j$. Since T_j reads-x-from T_i in H, $T_i \rightarrow T_j$ is in the RDSG of H, so T_i precedes T_j in H_s. Since the RDSG induces both a read and a write order, we have that either $T_k \ll T_i$ or $T_j \ll T_k$. Thus either T_k precedes T_i (which precedes T_j) or T_k follows T_j in H_s, both contradicting that T_j reads-x-from T_k in H_s.

Now assume T_j reads-x-from T_i in H_s. By conditions (3) and (4) in the definition of RD history and the definition of reads-from, T_j reads-x-from some transaction in H, say T_h. By the previous paragraph, T_j reads-x-from T_h in H_s. Since read-from relationships are unique, $T_h = T_i$. □

8.5 ATOMICITY OF FAILURES AND RECOVERIES

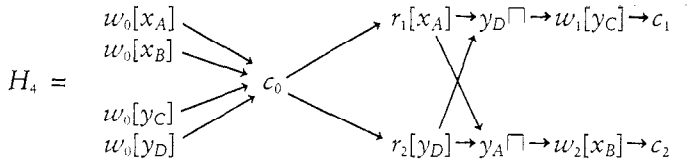
Another characterization of 1SR histories is that they are RD histories in which failure and recovery events appear to be atomic. That is, in a 1SR history, all transactions have a consistent view of when copies fail and recover. In the next

two subsections, we will explain this characterization by means of examples. Then we will describe a graph structure that captures this characterization.

Atomicity of Failures

Loosely speaking, a transaction learns about a failure when it tries to read or write a copy that turns out to be unavailable. Similarly, it knows that a copy could not have failed yet if it successfully accesses that copy. Depending on when each transaction learns about failures, different transactions might see failures occurring in different orders.

For example, reconsider history H_4.



Since T_1 decided not to write y_D, it must have learned about y_D's failure before it committed; this means that in an equivalent serial execution, y_D failed before T_1 executed. And since it read x_A, it believes x_A failed after it executed. Thus, T_1 sees failures in the following order: $y_D\sqcap \rightarrow T_1 \rightarrow x_A\sqcap$. By contrast, T_2 sees the failures in the opposite order. It didn't write x_A, so it must have learned that x_A failed before it ran. But it read y_D, so it believes y_D failed after it ran. Thus, it sees $x_A\sqcap \rightarrow T_2 \rightarrow y_D\sqcap$.

Suppose we think of failures as atomic events that must be recorded in a serial execution. Given T_1's and T_2's view of these events, there is no serial execution of T_1 and T_2 in which we can place the failure events $x_A\sqcap$ and $y_D\sqcap$. For example, if the given execution were equivalent to the serial execution T_1T_2, then $T_1 \rightarrow x_A\sqcap \rightarrow T_2$ is consistent with T_1's and T_2's view. But T_1 thinks y_D failed before T_1, and T_2 thinks y_D failed after T_2. Both views cannot be true if $y_D\sqcap$ is an atomic event. Instead, if the given execution were equivalent to T_2T_1, then we have $T_2 \rightarrow y_D\sqcap \rightarrow T_1$, but now we have inconsistent views of $x_A\sqcap$ to contend with.

To ensure that failure events appear to be atomic, we need to synchronize failures with Reads and Writes. Unfortunately, failures are not controllable events. They happen whenever they want to. So, the best we can do is ensure that transactions *see* failures in a consistent order. This, as it turns out, is all we need.

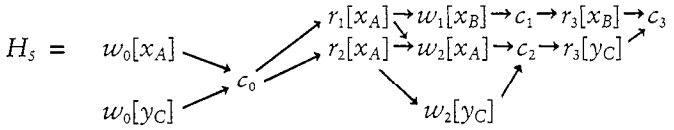
Note that a transaction can see a failure by *not* issuing a Write on an unavailable copy. Therefore, the DBS must not only control the order in which transactions read and write copies, but must also control the transaction's decision whether or not to issue a Write on a copy. Thus, the TM's translation activity has an effect on one-copy serializability.

Atomicity of Recoveries

To create a new copy of x, or to recover a formerly failed copy of x, say x_B, the DBS must store an initial value in x_B. Any transaction that ordinarily writes into x can initialize x_B. Whenever it writes into other copies of x, it writes into x_B too. Rather than waiting for such a transaction to appear, the DBS can force the initialization of x_B by running a special transaction, called a *copier*. The copier simply reads an existing copy of the data item that is up-to-date and writes that value into the new copy. Whether the DBS uses an ordinary transaction or a copier to initialize x_B, it must ensure that no transaction reads x_B until it has been initialized.

The DBS must also make x_B known to all transactions that update x, so that they will write into x_B whenever they write into other copies of x. This latter activity requires some synchronization, as the following example illustrates.

Suppose the database has copies x_A and y_C at the time that the DBS is ready to create copy x_B. It runs a copier transaction, T_1, to initialize x_B at about the same time that two other transactions T_2 and T_3 execute. The resulting execution is as follows.



H_5 is incorrect. The only serial RD history equivalent to H_5 is

$$H_5' = w_0[x_A]\ w_0[y_C]\ c_0\ r_1[x_A])]\ w_1[x_B]\ c_1\ r_2[x_A]\ w_2[x_A]\ w_2[y_C]\ c_2\ r_3[x_B]\ r_3[y_C]\ c_3$$

H_5' is not equivalent to the serial 1C history $T_0\ T_1\ T_2\ T_3$. In a one-copy database, T_2 would write x and y, and T_3 would read the values that T_2 wrote. But in this execution, T_3 read the value of x that T_1 wrote and the value of y that T_2 wrote. H_5 is not equivalent to any other serial 1C history either, and so is not 1SR.

The problem is that T_2 should have updated the new copy of x, x_B, but didn't. Since T_1 knows that x_B exists (it wrote x_B), and since T_1 effectively executes before T_2 (because $r_1[x_A] < w_2[x_A]$), T_2 should also know that x_B exists. Therefore, since T_2 writes into x, it should write all copies of x, including x_B. If it did, then $r_3[x_B]$ would have read the proper value.

We can also explain this in terms of the atomicity of recovery events. Let us denote the recovery (or initialization) of x_A as $x_A\sqcup$. Since T_1 wrote x_B, it believes x_B recovered before it executed. That is, $x_B\sqcup \rightarrow T_1$. Since T_2 wrote x_A but not x_B, it believes x_B recovered after it executed. That is, $T_2 \rightarrow x_B\sqcup$. But since $T_1 \rightarrow T_2$ is in SG(H_5), these views of the recovery of x_B are inconsistent.[6]

[6]An alternative analysis is $x_B\sqcup \rightarrow T_1 \rightarrow x_B\sqcap \rightarrow T_2 \rightarrow x_B\sqcup \rightarrow T_3$. We have $x_B\sqcup \rightarrow T_3$ because T_3 read x_B. But now T_3 reads the recovered copy x_B before it has been initialized (with the value of x written by T_2), which is illegal.

*Failure-Recovery Serialization Graphs

As we have seen, another explanation why a serializable execution may not be 1SR is that different transactions observe failures and recoveries in different orders. We can formalize this reasoning by augmenting SGs to include nodes that represent the creation and failure of each copy. To keep the notation simple, we will assume that each copy is created once and sometime later fails. *Once it has failed, a copy never recovers.* This assumption is not a loss of generality because we can regard the recovery of a copy as the creation of a new copy. That is, a copy that fails and recovers several times is represented by a sequence of uniquely named incarnations, each of which is created, fails at most once, and never recovers. An alternative model for incarnations is suggested in Exercise 8.9.

Given an RD history H over transactions $\{T_0, \ldots, T_n\}$, a *failure-recovery serialization graph* (*FRSG*) for H is a directed graph with nodes N and edges E where:

$$N = \{T_0, \ldots, T_n\} \cup \{\text{create}[x_A] \mid x \text{ is a data item and } x_A \text{ is a copy of } x\} \cup \{\text{fail}[x_A] \mid x \text{ is a data item and } x_A \text{ is a copy of } x\}$$

$$E = \{T_i \rightarrow T_j \mid T_i \rightarrow T_j \text{ is an edge of SG}(H)\} \cup E1 \cup E2 \cup E3$$

where

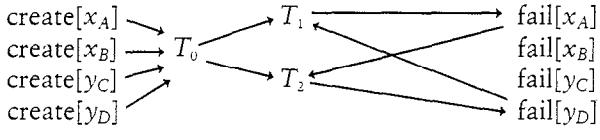
$$E1 = \{\text{create}[x_A] \rightarrow T_i \mid T_i \text{ reads or writes } x_A\}$$

$$E2 = \{T_i \rightarrow \text{fail}[x_A] \mid T_i \text{ reads } x_A\}$$

$$E3 = \{T_i \rightarrow \text{create}[x_A] \text{ or } \text{fail}[x_A] \rightarrow T_i \mid T_i \text{ writes some copy of } x, \text{ but not } x_A\}$$

The edges in $E1$ signify that if T_i read or wrote x_A, then x_A was created before T_i executed ($\text{create}[x_A] \rightarrow T_i$). $E2$ signifies that if T_i read x_A, then T_i executed before x_A failed ($T_i \rightarrow \text{fail}[x_A]$). Notice that this need not hold if T_i wrote (but did not read) x_A. $E3$ signifies that if T_i wrote some copies of x but did not write x_A, then it must have written those copies at a time when x_A did not exist, that is, either before it was created or after it failed. Thus, for each such situation, $E3$ must contain either $T_i \rightarrow \text{create}[x_A]$ or $\text{fail}[x_A] \rightarrow T_i$. As in RDSGs, H does not uniquely determine an FRSG.

For example, the following is an FRSG for H_4.



Since T_1 wrote y_C, but not y_D, we must include either $T_1 \rightarrow \text{create}[y_D]$ or $\text{fail}[y_D] \rightarrow T_1$; we chose the latter. Similarly, we added $\text{fail}[x_A] \rightarrow T_2$, because T_2 wrote x_B but not x_A. Notice that this FRSG has a cycle: $T_1 \rightarrow \text{fail}[x_A] \rightarrow T_2 \rightarrow \text{fail}[y_D] \rightarrow T_1$. In fact, *every* FRSG for H_4 has a cycle.

Given a history H, if there is an acyclic FRSG(H), then we can produce a serial history that is equivalent to H and which includes the creation and failure of copies as atomic events. That is, all transactions observe the creation and failure of each copy in the same order. This condition is enough to show that H is 1SR.

Theorem 8.5: Let H be an RD history. If H has an acyclic FRSG, then H is 1SR.

Proof: As in Theorem 8.4, let $H_s = T_{i_1} T_{i_2} \ldots T_{i_n}$ be a serial 1C history where $T_{i_1}, T_{i_2}, \ldots, T_{i_n}$ is a topological sort of FRSG(H). Since FRSG(H) contains SG(H), by Theorem 8.3 we can prove that H is 1SR just by proving that H and H_s have the same reads-from relationships.

First assume T_j reads-x_A-from T_i in H. Hence $T_i \rightarrow T_j$ is in the FRSG and T_i precedes T_j in H_s. Let T_k be any other transaction that writes x. If T_k writes x_A, then since T_j reads-x_A-from T_i, either $T_k \rightarrow T_i$ or $T_j \rightarrow T_k$ must be in the FRSG. If T_k does not write x_A, by definition of FRSG, either $T_k \rightarrow \text{create}[x_A]$ or $\text{fail}[x_A] \rightarrow T_k$. In the former case, since $\text{create}[x_A] \rightarrow T_i$, T_k precedes T_i in the FRSG. In the latter case, since $T_j \rightarrow \text{fail}[x_A]$, T_j precedes T_k in the FRSG. Hence, if T_k writes x, either T_k precedes T_i or follows T_j in the FRSG and in H_s, too. Thus, T_j reads-x-from T_i in H_s.

Now, suppose T_j reads-x-from T_i in H_s. By the definition of RD history, T_j reads-x-from some transaction in H, say T_h. By the previous paragraph, T_j reads-x-from T_h in H_s. Since reads-from relationships are unique, $T_h = T_i$. □

8.6 AN AVAILABLE COPIES ALGORITHM

Available copies algorithms handle replicated data by using enhanced forms of the write-all-available approach. That is, every Read(x) is translated into a Read of *any* copy of x and every Write(x) is translated into Writes of all available copies of x. Simply doing this is not enough to guarantee one-copy serializability, as history H_4 showed. Available copy algorithms enforce special protocols that, in conjunction with this "read-any, write-all-available" discipline, ensure correctness.

These algorithms handle site failures but not communications failures. That is, they assume that every site is either operational or down, and that all operational sites can communicate with each other. Therefore, each operational site can independently determine which sites are down, simply by attempting to communicate with them. If a site doesn't respond to a message within the timeout period, then it must be down.

For available copies algorithms, we'll assume that *the scheduler uses strict two phase locking*. Thus, after transaction T_i has read or written a copy of x_A, no other transaction can access x_A in a conflicting mode until after T_i has committed or aborted.

In this section we'll describe a simple available copies algorithm. We assume that there is a fixed set of copies for each data item, known to every site. This set does not change dynamically. Moreover, to keep the description simple, we'll initially assume that each copy is created and fails at most once. Later we'll discuss how to accommodate repeated failures and recoveries of copies — no changes to the algorithm are needed for this extension! After a copy has been initialized and before it has failed, it is said to be *available*; otherwise, it is *unavailable*.

Processing Reads and Writes

When a transaction T_i issues a Read(x), the TM at T_i's home site[7] (henceforth simply "T_i's TM") will select some copy x_A of x and submit Read(x_A) on behalf of T_i to site A. Typically the selected copy will be the one "closest" to T_i's home site (ideally the home site itself), to minimize the communication cost incurred by the Read. The correctness of the algorithm, however, does not depend on which copy is read — any one will do.

Site A regards x_A as *initialized* if it has already processed a Write on x_A, even if the transaction that issued the Write has not yet committed. If site A is operational and x_A is initialized, Read(x_A) will be processed by the scheduler and DM at A. If x_A is initialized but the transaction T_j that initialized x_A has not yet committed, then by Strict 2PL the scheduler must delay Read(x_A) until T_j commits or aborts. If Read(x_A) is rejected, a negative acknowledgment will be returned and T_i will be aborted.[8] If Read(x_A) is accepted, the value read will be returned to T_i's TM and Read(x) will be done. However, if site A is down or if x_A hasn't been initialized, then T_i's TM will eventually time out while waiting for a response. In that event the TM could abort T_i or, better, it could submit Read(x_B) to another site B that contains a copy of x. As long as one of the copies of x can be read, Read(x) will be successful. If no copy of x can be read, T_i must abort.

When T_i issues a Write(x) operation, its TM sends Write(x_A) operations to every site A where a copy of x is supposed to be stored. If A is down, Write(x_A) will not be received and, of course, its processing will never be acknowledged. T_i's TM will eventually time out waiting for a response from A. If A is operational, the handling of Write(x_A) depends on whether or not x_A was previously initialized.

If x_A has been initialized, then Write(x_A) must be processed by A's scheduler and DM, and eventually a response is returned to T_i's TM indicating whether Write(x_A) was rejected or duly processed.

---

[7]Recall from Section 7.1 that T_i's "home site" is the site where it originated and whose TM supervises its execution.

[8]It is pointless to submit a Read(x_B) for a different copy x_B of x, as the conflict that caused the rejection of Read(x_A) will also cause the rejection of Read(x_B).

On the other hand, if x_A has not been initialized, the DBS at A has two options. It could use Write(x_A) to initialize x_A at this time (in which case the operation is processed as just described), or it may ignore Write(x_A), preferring to not initialize x_A yet. Ignoring the operation means that A doesn't send an acknowledgment to T_i's TM, that is, A acts as if it were down as far as x_A is concerned.[9]

After sending Writes to all of x's copies, T_i's TM waits for responses. It may receive rejections from some sites, positive responses from others (meaning the Write has been accepted and performed), and no responses from others (those that have failed or that have not initialized their copies of x). Writes for which no responses are received are called *missing writes*. If *any* rejection is received or if *all* Writes to x's copies are missing, then Write(x) is rejected and T_i must abort. Otherwise, Write(x) is successful.

Validation

So far we've described, in some detail, the "read-one, write-all-available" discipline. We know from history H_4 that this isn't enough to guarantee one-copy serializability. The available copies algorithm uses a *validation protocol* to ensure correctness. Transaction T_i's validation protocol starts after T_i's Reads and Writes on copies have been acknowledged or timed out. At that time T_i knows all its missing writes as well as all the copies it has actually accessed (read or written). The validation protocol consists of two steps:

1. *missing writes validation*, during which T_i makes sure that all copies it tried to, but couldn't, write are still unavailable, and
2. *access validation*, during which T_i makes sure that all copies it read or wrote are still available.

It is important that missing writes validation be performed before access validation (see Exercise 8.11).

To validate missing writes, T_i sends a message UNAVAILABLE(x_A) to site A, for each copy x_A that T_i found unavailable. A will acknowledge such a message only if it has, in the meanwhile, initialized x_A (i.e., received and processed a Write(x_A), even if the Write has not yet been committed).

After sending the UNAVAILABLE messages, T_i waits for responses. At the end of the timeout period, if it has *not* received *any* acknowledgments to these messages, it proceeds with access validation. Otherwise, some copy it hasn't updated has been initialized and T_i is aborted. At this point it is important to recall the assumption that there are no communication failures. It implies that if T_i has not received any acknowledgment to the UNAVAILABLE messages by

[9]Alternatively, A might send a message expressly indicating that x_A has not been initialized yet. This will prevent T_i's TM from waiting for the full timeout period before concluding that x_A is not available.

the end of the timeout period, then it must be that no such acknowledgment was sent and, therefore, that all copies T_i couldn't write are still unavailable.[10]

If the missing writes validation step succeeds, then T_i proceeds with access validation. To that end, T_i sends a message AVAILABLE(x_A) to site A, for each copy x_A that T_i read or wrote. A acknowledges this message if x_A is still available at the time A receives the message. If *all* AVAILABLE messages are acknowledged, then access validation succeeds and T_i is allowed to commit. Otherwise T_i must abort.

The validation protocol requires a significant amount of communication and is therefore expensive. We can reduce the number of messages sent by combining all UNAVAILABLE messages from T_i to some particular site into one message. Similarly for AVAILABLE messages and acknowledgments. Even so, we will have two steps (missing writes validation and access validation), each requiring two rounds of message exchanges (one for the UNAVAILABLE/AVAILABLE messages and one for acknowledgments). Fortunately, things aren't as bad as this may suggest.

First, if a transaction has no missing writes, there is no need for missing writes validation! Thus, after all copies have been initialized and in the absence of (site) failures, the first step of the validation protocol is avoided.

Second, access validation can be combined with atomic commitment. Recall that when T_i terminates, its TM sends VOTE-REQ messages to all sites where T_i accessed copies. The VOTE-REQ message sent to a site can be used as an implicit AVAILABLE message. If a site responds YES, surely all copies accessed by T_i at that site are still available; thus a YES response can be used as an implicit acknowledgment for AVAILABLE. Therefore, access validation can ride for free on the coattails of atomic commitment.[11]

Some Examples

To develop some insight on how the validation protocol ensures one-copy serializability, we consider some examples of non-1SR executions and show how they are prevented from happening. In the next subsection we prove the correctness of the algorithm.

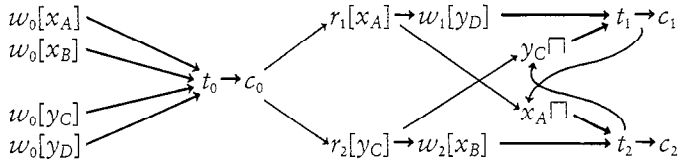
Let's start with history H_4. We reproduce it next, embellished with certain new symbols to represent events of interest. For transaction T_i, t_i represents the

[10]It is possible that a copy was initialized after T_i attempted to write it and failed before T_i started its missing writes validation step. The proof of the algorithm's correctness, which will be presented shortly, should quell any concern that such behavior could compromise one-copy serializability.

[11]Under certain circumstances the atomic commitment protocol needn't involve sites where a transaction only read but did not write (cf. Section 7.3). However, access validation requires that messages be sent to such sites, to verify that copies read are still available. Thus, some additional cost to atomic commitment may be incurred by access validation.

moment when T_i *begins* its access validation step. By the specification of the algorithm, t_i must follow all Read and Write operations of T_i as well as the missing writes validation step (if present), and must precede c_i. As usual, the symbol $x_A\sqcap$ stands for the failure of x_A.

H_4' =



Consider the precedences involving the failure and access validation events. Since T_1 read x_A, the failure of x_A must have occurred after T_1 started its access validation. Otherwise T_1 would have found x_A to be unavailable and would therefore abort. Therefore, $t_1 < x_A\sqcap$ and, similarly, $t_2 < y_C\sqcap$.

Since T_1 wrote y_D but not y_C, it must have carried out missing writes validation and found that y_C is still unavailable. Since at that time y_C had already been initialized, it must be that y_C failed before the completion of T_1's missing writes validation and thus before the beginning of access validation. Hence, $y_C\sqcap < t_1$. The precedence $x_A\sqcap < t_2$ is justified on similar grounds.

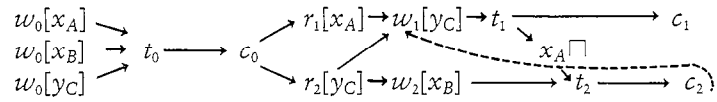
Given these precedences we have a cycle in H_4': $t_1 < x_A\sqcap < t_2 < y_C\sqcap < t_1$. This is impossible since H_4', being a history, is supposed to be a partial order. This means that H_4' could not have happened.

In terms of the algorithm, the reason why H_4' couldn't have happened is this: If T_1's access validation succeeded we'd have $t_1 < x_A\sqcap < t_2$, so T_2's access validation started after T_1's. Since $y_C\sqcap < t_1$, we get $y_C\sqcap < t_2$. But then T_2's access validation would not have succeeded, so T_2 would have aborted. Similarly, if T_2's validation succeeded, T_1's would fail. (Of course, both transactions' validations could have failed.)

Note the pivotal role of validation in the justification of why H_4' couldn't occur. Without missing writes and access validation, we wouldn't be able to assert the existence of the precedences that yield the cycle in H_4'.

Our next example illustrates the significance of the assumption that the scheduler uses Strict 2PL. History H_6 is an execution of T_1, T_2 (the same transactions as in H_4) where there is only one copy of y, y_C. For the moment ignore the broken arrow from c_2 to $w_1[y_C]$.

H_6 =



Like H_4, H_6 is not 1SR; T_1 doesn't read the value of x written by T_2, and T_2 doesn't read the value of y written by T_1, one of which would have to hold in any serial one-copy execution over T_1, T_2. Unfortunately, the validation steps don't prevent either T_1 or T_2 from committing, as you can easily verify. What stops H_6 from happening is the scheduler's use of Strict 2PL. Since T_2 read y_C

before T_1 wrote into it, T_2 must have locked y_C before T_1 did. But then T_1 won't lock y_C until after T_2 has committed; i.e., we have $c_2 < w_1[y_C]$ (a stronger precedence than just $r_2[y_C] < w_1[y_C]$). Now we get a cycle in H_6, meaning that H_6 couldn't have occurred. Note that the precedence $r_2[y_C] < w_1[y_C]$ isn't enough to give rise to a cycle. It's important that T_2 keeps its locks until it commits.

*Proof of Correctness

We'll state seven properties that are satisfied by histories representing executions of the available copies algorithm. We'll justify why each property is satisfied and then show that any history satisfying them is 1SR, thereby proving that the available copies algorithm is correct.

In addition to the symbols we used in the previous examples, we'll use the symbol t_i' to denote the point at which T_i begins its missing writes validation step. Hence all Read and Write operations of T_i must precede t_i' and $t_i' < t_i$. We'll also use the symbol $x_A\sqcup$ to denote the creation (initialization) of copy x_A. x_A is created when the first transaction that writes it, say T_0, begins its access validation. Thus $x_A\sqcup$ is a "synonym" for t_0.

The properties that must be satisfied by any history H produced by the available copies algorithm are as follows

AC_1: For every T_i, $t_i' < t_i < c_i$. And for every Read or Write p_i of T_i, $p_i < t_i'$.

AC_2: SG(H) is acyclic.

AC_3: If $T_i \rightarrow T_j$ is in SG(H) then $t_i < t_j'$.

AC_1 defines the execution and validation phases of a transaction. AC_2 is satisfied because the scheduler that controls access to copies produces serializable executions. AC_3 is satisfied because the scheduler uses Strict 2PL. If $T_i \rightarrow T_j$ is an edge in SG(H), then there exist conflicting operations $p_i[x_A] < q_j[x_A]$ in H. By Strict 2PL, $c_i < q_j[x_A]$. Consequently, $t_i < t_j'$ (because, by AC_1, $t_i < c_i$ and $q_j[x_A] < t_j'$).[12]

AC_4: For any $r_i[x_A]$ in H, $x_A\sqcup < r_i[x_A]$.

AC_5: For any $w_i[x_A]$ in H, either $x_A\sqcup = t_i$ or $x_A\sqcup < w_i[x_A]$.

AC_4 and AC_5 say that a copy must be initialized before it may be read or written. This is obviously satisfied by the available copies algorithm (cf. the subsection on Processing Reads and Writes).

AC_6: If $r_i[x_A]$ or $w_i[x_A]$ is in H then $t_i < x_A\sqcap$.

---

[12]At the beginning of this section we said that available copies algorithms require a Strict 2PL scheduler. That's not quite true. The correct statement is that these algorithms require a scheduler that will guarantee AC_2 through AC_5.

AC_6 says that T_i's access validation begins before the failure of any copy read or written by T_i. This is satisfied because otherwise one of T_i's AVAILABLE messages would not be acknowledged and so T_i would be aborted.

AC_7: If T_i writes into some copy of x but not x_A then either $x_A\sqcap < t_i$ or, for any $w_j[x_A]$ in H, $t_i' < w_j[x_A]$.

To understand AC_7, consider the missing writes validation step, in which T_i sent UNAVAILABLE(x_A) to site A. If A had responded to the message, then T_i would have aborted. So assume not. Thus, either x_A failed before A received the message (in which case $x_A\sqcap < t_i$) or x_A had not yet been initialized (in which case $t_i' < w_j[x_A]$ for all T_j that write x_A.

AC_7 is where we use the fact that missing writes validation completes before access validation begins. Without this handshake, the two validation steps would execute concurrently, so t_i and t_i' would not be distinguishable events. If x_A failed before A received UNAVAILABLE(x_A), we could conclude that $x_A\sqcap < c_i$, but not that $x_A\sqcap < t_i' = t_i$, which is needed in the correctness proof.

We begin the proof with a preliminary lemma that strengthens AC_7.

Lemma 8.6: Let H be a history produced by the available copies algorithm. In H, if T_i writes some copy of x but not x_A, then either $x_A\sqcap < t_i$ or $t_i < x_A\sqcup$.

Proof: Suppose T_i writes into x_B but not x_A. By AC_7 either $x_A\sqcap < t_i$ or t_i' $< w_0[x_A]$, where T_0 is the transaction that initialized x_A. In the former case, H obviously satisfies the lemma. Therefore let us assume the latter and consider two cases, depending on whether or not T_0 wrote into x_B.

CASE 1: Suppose $w_0[x_B]$ is in H. We claim that $w_i[x_B] < w_0[x_B]$. For, otherwise, we'd have $w_0[x_B] < w_i[x_B]$ and, by AC_3, $t_0 < t_i'$. By assumption, $t_i' < w_0[x_A]$. By AC_1, $w_0[x_A] < t_0$, so by transitivity $t_0 < t_0$, a contradiction. Therefore, $w_i[x_B] < w_0[x_B]$. By AC_3, $t_i < t_i'$. Since $t_0' < t_0$ by AC_1, and $t_0 = x_A\sqcup$ by assumption, we have $t_i < x_A\sqcup$, as desired. □

CASE 2: Suppose T_0 doesn't write x_B. By AC_7 either (a) $x_B\sqcap < t_0$ or (b) $t_0' < w_i[x_B]$. If (a), we have $t_i < x_B\sqcap$ (by AC_6), so $t_i < t_0 = x_A\sqcup$, and we get $t_i < x_A\sqcup$, as desired. If (b), we have $t_i' < w_0[x_A]$ by assumption, $w_0[x_A]$ $< t_0'$ by AC_1, $t_0' < w_i[x_B]$ (this is (b)), and $w_i[x_B] < t_i'$ by AC_1. All these imply $t_i' < t_i'$, a contradiction. □

Any history that satisfies AC_1 - AC_7 is an RD history. Conditions (1), (2), and (4) in the definition of RD history are immediate. Condition (3), which requires each $r_j[x_A]$ to be preceded by a $w_i[x_A]$, follows AC_4. Condition (5) says that if $w_i[x] <_i r_i[x]$ and $r_i[x_A] \in H$, then $w_i[x_A] \in H$. To see this, suppose $w_i[x_A]$ $\notin H$. Then by Lemma 8.6, either (a) $x_A\sqcap < t_i$ or (b) $t_i < x_A\sqcup$. AC_6 contradicts

(a). By AC_4 $x_A\sqcup < r_i[x_A]$, and by AC_1 $r_i[x_A] < t_i$; thus $x_A\sqcup < t_i$, which contradicts (b). Hence, $w_i[x_A] \in H$ as desired.

Theorem 8.7: The available copies algorithm produces only 1SR histories.

Proof: Let H be a history that satisfies AC_1 - AC_7. We claim H is 1SR. Since H is an RD history, by Theorem 8.5 it's enough to show that H has an acyclic FRSG. From H construct a graph G with nodes N and edges E, where:

$$\begin{aligned}
N &= \{T_i \mid T_i \text{ appears in } H\} \cup \\
&\quad \{\text{create}[x_A], \text{fail}[x_A] \mid x_A \text{ is a copy of some data item}\} \\
E &= \{T_i \rightarrow T_j \mid T_i \rightarrow T_j \text{ is an edge of SG}(H)\} \cup \text{F1} \cup \text{F2} \cup \text{F3} \cup \text{F4} \\
\text{F1} &= \{\text{create}[x_A] \rightarrow T_i \mid x_A\sqcup < t_i \text{ or } x_A\sqcup = t_i\} \\
\text{F2} &= \{T_i \rightarrow \text{fail}[x_A] \mid t_i < x_A\sqcap\} \\
\text{F3} &= \{\text{fail}[x_A] \rightarrow T_i \mid x_A\sqcap < t_i\} \\
\text{F4} &= \{T_i \rightarrow \text{create}[x_A] \mid t_i < x_A\sqcup\}
\end{aligned}$$

To show that G is an FRSG, we must show that it contains the edges E1–E3 in the definition of FRSG.

$$\begin{aligned}
\text{E1} &= \{\text{create}[x_A] \rightarrow T_i \mid T_i \text{ reads or writes } x_A\} \\
\text{E2} &= \{T_i \rightarrow \text{fail}[x_A] \mid T_i \text{ reads } x_A\} \\
\text{E3} &= \{T_i \rightarrow \text{create}[x_A] \text{ or } \text{fail}[x_A] \rightarrow T_i \mid T_i \text{ writes some copy of } x \\
&\quad \text{but not } x_A\}
\end{aligned}$$

By AC_6, if T_i reads x_A, then $t_i < x_A\sqcap$, so E2 $\subseteq$ F2. By Lemma 8.6, if T_i writes x but not x_A, then either $x_A\sqcap < t_i$ or $t_i < x_A\sqcup$, so some E3 is contained in F3 $\cup$ F4.

To show G contains E1, suppose T_i reads x_A. By AC_4, $x_A\sqcup < r_i[x_A]$, and by AC_1, $r_i[x_A] < t_i$. Hence, $x_A\sqcup < t_i$. If T_i writes x_A, then by AC_5, either $x_A\sqcup = t_i$ or $x_A\sqcup < w_i[x_A]$. In the latter case, by AC_1 $w_i[x_A] < t_i$, so $x_A\sqcup < t_i$. Thus E1 $\subseteq$ F1, and G contains an FRSG for H.

We now show that G is acyclic. Define a mapping f by $f(\text{create}[x_A]) = x_A\sqcup$, $f(\text{fail}[x_A]) = x_A\sqcap$, and $f(T_i) = t_i$. If $n_j \rightarrow n_k$ is in G, then either $f(n_j) < f(n_k)$ in H or $f(n_j) = f(n_k) = x_A\sqcup$ for some x_A (i.e., $n_j = \text{create}[x_A]$ and $n_k = t_i = \text{create}[x_A]$; see F1). Thus, for any cycle $P = n_1, \ldots, n_m, n_1$ in G ($m > 1$), there is a corresponding sequence $P' = f(n_1) \leq \ldots \leq f(n_m) \leq f(n_1)$ in H. Clearly, P must contain at least one edge $n_j \rightarrow n_k$ not in F1. For each such edge, $f(n_j) < f(n_k)$, by definition of F2 - F4. Therefore, $f(n_1) < f(n_1)$ in P, contradicting that H is a partial order. Thus, G is acyclic.

Since G contains an FRSG for H, that FRSG is acyclic too. Thus, by Theorem 8.5, H is 1SR. □

Repeated Failures and Recoveries

So far we have assumed that each copy is initialized and fails at most once. This assumption results in a notational simplification — that for each copy x_A, we have at most one $x_A\sqcup$ and one $x_A\sqcap$ symbol. The usual justification for this assumption is that one can view a single copy that's initialized and fails repeatedly as a sequence of copies, each of which is created and fails once.

This view might appear to contradict another basic assumption of the available copies algorithm, namely, that the set of copies is fixed. Fortunately, as long as *all* copies of a data item don't fail, the "contradiction" is a red herring (see Exercise 8.13). Though the set of copies is fixed, it does not have to be finite!

In particular, for each site A where there is a copy of x we can imagine we have an infinite supply of such copies, x_A^1, x_A^2, x_A^3, ... At any time at most one of these copies is available. If copy x_A^i is the one presently available, then copies x_A^1, ..., x_A^{i-1} have failed and copies x_A^{i+1}, x_A^{i+2}, ... are uninitialized. When (and if) x_A^i fails, x_A^{i+1} will be the next copy of x at site A to be initialized. Since there is a finite number of sites and at most one copy of x is available at any one of them, Write(x) results in the writing of a finite number of copies — a relief! Read(x) involves just one copy, so there is no problem here. Access validation involves the copies read or written — a finite number, as we just argued.

Missing writes validation, however, must ensure that an infinite number of copies is unavailable. Since these copies are stored at a finite number of sites, we can do this with a finite number of messages. The only problem is that now any transaction that writes some data item x will have missing writes (in fact an infinite number of them!), *even if it writes copies at all sites where x is stored*. Does this mean that every transaction must perform missing writes validation? If this were so, it would negate our earlier assertion that in the absence of failures the available copies algorithm incurs no validation cost at all. Fortunately, it isn't so. The reason is that access validation can now be used implicitly for missing writes validation. That is, if T_j writes copy x_A^i and during access validation it finds x_A^i is still available, it implicitly knows that copies x_A^1, ..., x_A^{i-1}, x_A^{i+1}, x_A^{i+2}, ... are still unavailable. Thus there is no need for separate missing writes validation in this case. Of course, missing writes validation is required, as before, if T_j couldn't write any copy of x at site A.

8.7 DIRECTORY-ORIENTED AVAILABLE COPIES

The static assignment of copies to sites in the available copies algorithm of the previous section is a serious disadvantage. It requires, among other things, that transactions attempt to update copies at down sites. If site failures persist for

long periods, this is clearly inefficient. In addition, it is not possible to dynamically create or destroy copies at new sites.

In this section we study the *directory-oriented* available copies algorithm, which rectifies these problems. The algorithm uses directories to define the set of sites that currently stores the copies of an item. More precisely, for each data item x there is a directory $d(x)$ listing the set of x's copies. Like a data item, a directory may be replicated, that is, it may be implemented as a set of directory *copies*, stored at different sites.

The directory for x at site U, denoted $d_U(x)$, contains a list of the copies of x and a list of the directory copies for x that site U believes are available. For notational clarity, we will typically use $\{U, V\}$ for names of sites that store directories, and $\{A, B, C, D\}$ for those that store data item copies. These sets of sites need not be disjoint. Usually, a site will store both directory and data item copies. We assume that before we begin executing transactions, all copies of directories exist, but that no copies of data items exist. That is, we assume that new directory copies are never created. A method for creating directory copies appears in a later subsection.

Directories are treated like ordinary data items by the DBS. In particular, concurrent access to directory copies is controlled by the same scheduler that controls concurrent access to data item copies.[13] The only difference is that ordinary transactions can only read directories. Directories are updated by two special transactions, *Include* (or *IN*) for creating new data item copies and *Exclude* (or *EX*) for destroying data item copies.

Basic Algorithm

When a site A containing x recovers from failure, or when A wants to create a new copy of x, the DBS runs a transaction $\text{IN}(x_A)$. $\text{IN}(x_A)$ brings the value of x_A up-to-date by:

1. finding a directory copy $d_U(x)$, for example, by using a local copy that it knows exists, by reading a master directory that lists copies of $d(x)$, or by polling other sites;
2. reading $d_U(x)$ to find an available copy of x, say x_B;
3. reading x_B; and
4. copying x_B's value into x_A.

Thus, it performs the function of a copier. If $d_U(x)$ says that there are no copies of x and never were any, then A should provide an initial value for x_A, the first

[13] As for the simple available copies algorithm, we are assuming a Strict 2PL scheduler or, to be somewhat more general, that transactions that access a data item copy in conflicting modes start their validation protocols in the order in which they accessed that copy.

copy of x.[14] In any case, it declares x_A to be available by adding x_A to each available copy of $d(x)$.

When a site fails, some DBS that tries to access data at that site observes the failure. Based on directories it has read, the DBS believes that certain copies are stored at the failed site. Therefore, for each such copy, it runs an EX transaction for each copy stored there. EX(x_A) declares x_A to be unavailable by removing x_A from every available copy of $d(x)$. That is, it

1. reads some directory copy $d_U(x)$,
2. removes x_A from the list of available copies it read from $d_U(x)$, and
3. writes that updated list into each directory copy listed in $d_U(x)$.

Notice that if a DBS incorrectly believes a copy x_A was at the failed site when in fact it wasn't, it does no harm by executing EX(x_A).

Let NX be any IN or EX transaction. NX(x_A) begins by reading a directory, say $d_U(x)$. It expects to be able to update all of $d(x)$'s copies listed in $d_U(x)$. However, some of these directory copies may have recently failed and are therefore unavailable. Not only is NX unable to update such directory copies, but it now knows that the list of available directories in the (remaining) available copies of $d(x)$ directories is wrong. NX corrects such errors as follows.

After reading $d_U(x)$, NX attempts to access the directory copies listed in $d_U(x)$. Let AD be the directory copies listed in $d_U(x)$ that it determines are available (including $d_U(x)$). NX then updates each directory copy in AD by modifying its list of available data item copies *and* updating its list of available directory copies to be AD. This distribution of directory updates can be overlapped with the first phase of its ACP, by sending the directory update and VOTE-REQ in the same message.

To process Read(x) on behalf of a user transaction, the DBS reads a copy of $d(x)$, say $d_U(x)$. If it tries to read a directory copy that is unavailable, then it simply ignores the attempt and keeps trying other copies until it finds one that *is* available. It then selects a copy x_A of x that $d_U(x)$ says is available and issues Read(x_A). Since a failure can happen at any time, it is possible that $d_U(x)$ says that x_A is available, but the DBS discovers that x_A is unavailable when it tries to read it. In this case, the DBS can take one of three actions: (1) it can select another copy that $d_U(x)$ says is available and try to read that copy; (2) it can read a different copy of $d(x)$, $d_V(x)$, and try to read a copy that $d_V(x)$ says is available; or (3) it can abort the transaction.

To process Write(x), the DBS reads a copy of $d(x)$, $d_U(x)$, and issues Write(x_A) for every copy x_A that $d_U(x)$ says is available. If the DBS discovers

[14] If $d_U(x)$ says that there are no copies of x but that there *were* some in the past, then x is recovering from a total failure. In this case, A can initialize x using its last committed value for x only if x_A was among the last copies of x to have failed. It can determine this fact using the technique of Section 7.5. For pedagogical clarity, we will not incorporate this complexity in the remainder of this section. See Exercise 8.13).

that any copy that $d_U(x)$ says is available is actually unavailable, it must abort the transaction T_i that issued the Write and run an EX(x_A) transaction. When that commits, it can try running T_i again.[15]

After performing all its operations, transaction T_i must carry out its validation protocol. Since there are never missing writes, missing writes validation is unnecessary. The reason there are no missing writes is that the set of available copies for x is listed in $d(x)$ and, as we just saw, unless all these copies are actually written, T_i will abort. Thus only access validation is needed in the directory-oriented available copies algorithm (another advantage over the simpler algorithm of the preceding section). As we have seen, access validation can be done together with atomic commitment.

In fact, in the directory-oriented algorithm, access validation can be done merely by checking that the *directory* copies that T_i read still contain the data item copies T_i accessed (see Exercise 8.15). Thus, if directory copies are stored at all sites, access validation requires no communication at all.[16]

This is especially important if the DBS uses deferred Writes. The DBS cannot do Read validation until the transaction terminates, which doesn't occur until all Writes are distributed. If the DBS at site A can validate T_i only using directory copies stored at A, then it can do access validation locally. This avoids another round of communication after the distribution of T_i's Writes.

Correctness Argument

We can argue that this algorithm produces 1SR executions by using Theorem 8.5. The theorem says that if all transactions observe creations and failures in the same order, then the execution is 1SR. We can see that every execution of the algorithm satisfies this property by the following intuitive line of reasoning. We will show in a moment that user transactions behave as if the creation or failure of a copy occurred at the moment its IN or EX executed. All transactions, including INs and EXes, effectively execute in the serialization order. Thus, all user transactions see the same order of INs and EXes, and hence see the same order of creations and failures. This is the condition of Theorem 8.5, so the execution is 1SR.

A transaction T_i demonstrates its belief that a copy x_A is available by operating on x_A. If it operated on x_A then it must have read some directory copy $d_U(x)$ that said that x_A was available. IN(x_A) must have written x_A into $d_U(x)$

---

[15]A fancier (and better) way to do this is to have T_i incorporate EX(x_A)'s actions before proceeding. More precisely, T_i will execute EX(x_A) as a subtransaction and will commit only if EX(x_A) commits. Since this gets us into nested transactions, a topic not discussed in this book, we'll stick with the brute force method of aborting T_i, running EX(x_A), and then restarting T_i from scratch.

[16]Of course, the atomic commitment protocol must still be carried out. However, now it need not involve sites at which the transaction only read copies.

before T_i read $d_U(x)$. Thus, T_i executed after IN(x_A). Furthermore, after T_i terminated it checked that x_A was still available. Since EX(x_A) doesn't execute until after x_A fails, T_i executed before EX(x_A). So T_i executed after IN(x_A) and before EX(x_A). This is consistent with the view that x_A was created when IN(x_A) executed and failed when EX(x_A) executed.

A transaction T_i demonstrates its belief that a copy x_A is *not* available by *not* operating on that copy when it should, namely when it writes into data item x. When T_i writes x, it decides which copies to write by reading one of x's directories, say $d_U(x)$, and writing into all of the copies listed there. If x_A wasn't in $d_U(x)$ at the time the transaction read it, then T_i does not write x_A, thereby observing the failure of x_A. One way this could occur is if T_i read $d_U(x)$ before IN(x_A) wrote x_A into $d_U(x)$. The other possibility is that EX(x_A) wrote $d_U(x)$ to delete x_A before T_i read $d_U(x)$. In the former case, T_i executed before IN(x_A) and in the latter case it executed after EX(x_A). This too is consistent with the view that the copy was created when IN executed and failed when EX executed. Thus, transactions behave as if each IN (or EX) executes at the moment the copy is created (or fails), so, by Theorem 8.5, the execution is 1SR. This argument can be formalized along the lines of the proof given in the previous section for the simpler AC algorithm (see Exercise 8.16).

Creating Directories

To create a directory copy $d_U(x)$, we must store the current list of copies of x and directory copies for x in $d_U(x)$. We must also add $d_U(x)$ to the list of directories in every other directory for x. Thus, a *Directory Include* (or *DIN*) transaction for $d_U(x)$ begins by reading an existing directory copy for x, $d_V(x)$. Like an IN or EX, it checks to see which directory copies, AD, in $d_V(x)$ are still available. Then it updates the directory copies. It sets the list of data item copies in $d_U(x)$ to be that of $d_V(x)$. And it writes AD into the list of directory copies in each directory copy in AD (including $d_U(x)$).

How does a DIN know where to look for an existing copy of $d(x)$? If there is a copy of $d(x)$ at the site that is executing the DIN, then it can easily find the copy. If not, then it could poll other sites, to find out which of them have a copy.

If it can't find a copy, then it should try to initialize the first copy of $d(x)$. Unfortunately, it can't assume that it is the only DIN that is trying to initialize $d(x)$. The reason is that some other DIN may have been polling sites at the same time and reached the same conclusion that $d(x)$ was uninitialized. If both DINs try to create the initial copy of $d(x)$, neither copy will include a reference to the other. This could ultimately lead to an inconsistent state of x. For example, each subsequent IN would include copies of x that were only listed in one directory copy or the other. Then a user transaction would update only some copies of x, namely, those listed in the directory copy it read. It is therefore essential that only one DIN initialize a first copy of $d(x)$.

Another bad outcome will occur if $d(x)$ doesn't currently exist because it has experienced a total failure. That is, there used to be copies of $d(x)$, but they have all failed. It would be incorrect to initialize a new copy of $d(x)$, because there may still be copies of x around that are listed in the last copies of $d(x)$ to have failed.

One way to ensure that a DIN can correctly initialize the first copy of a directory is to have a master directory that contains a list of all directories for which an initial directory has been created and initialized. A DIN that initializes a directory $d(x)$ must update the master directory. If two DINs try to initialize $d(x)$ concurrently, they will have conflicting accesses to the master directory, so one DIN will have to wait until the other terminates. The second DIN will therefore see that it is not really the initial DIN for $d(x)$, and will act accordingly. Moreover, a total failure of $d(x)$ will be detectable, because the master directory will say that $d(x)$ exists, but no available site will have a copy.

Clearly, if the master directory is unavailable, then no directories can be initialized. The master directory should therefore be replicated, and all copies updated by each initial DIN. To avoid unbounded recursion, some additional restrictions are needed, such as fixing the set of master directory copies for all time. We leave the algorithms for including and excluding copies of the master directory as an exercise (see Exercise 8.18).

8.8 COMMUNICATION FAILURES

The main problem with available copies algorithms is that they do not tolerate communication failures. That is, if two or more operational sites cannot communicate, they may produce executions that are not 1SR.

For example, suppose there are copies $\{x_A, x_B, x_C\}$ of x and $\{y_B, y_C, y_D\}$ of y, but a communication failure has partitioned the network into two independent components, $P_1 = \{A, B\}$ and $P_2 = \{C, D\}$. That is, component P_1 has copies $\{x_A, x_B, y_B\}$ and P_2 has copies $\{x_C, y_C, y_D\}$. Suppose transactions T_1 and T_2 execute in components P_1 and P_2 (respectively) using an available copies algorithm, producing history H_7.

$$H_7 = w_0[x_A]\ w_0[x_B]\ w_0[x_C]\ w_0[y_B]\ w_0[y_C]\ w_0[y_D]\ c_0\ r_1[y_B] \rightarrow w_1[x_A] \rightarrow w_1[x_B] \rightarrow c_1\ r_2[x_C] \rightarrow w_2[y_C] \rightarrow w_2[y_D] \rightarrow c_2$$

In P_1, T_1 reads the one and only available copy of y, namely y_B, and writes both available copies of x, x_A and x_B. Similarly for T_2 in P_2. Since each transaction can validate that the copies it accessed were still available at the time it

terminated, each can commit. Yet the execution is clearly not 1SR, since neither transaction reads the output of the other.

The general problem is that transactions in different components may have conflicting accesses on different copies of the same data item, as in H_7 just shown. The serialization order of such transactions is essential to the one-copy serializability of their execution (cf. Section 8.4 and Theorem 8.4). But since DBSs executing in different components cannot communicate, they cannot synchronize transactions that execute in different components, and thus cannot ensure one-copy serializability. Therefore, nearly all techniques for handling communications failures focus on preventing transactions that access the same data item from executing in different components.[17]

Site Quorums

Surely one way to prevent conflicting transactions from executing in different components is to insist that only one component be allowed to process any transactions at all. Since the components can't communicate, each component must be able to independently decide whether *it* is the component that can process transactions. One way to accomplish this is to use *site quorums*.

In site quorums, we assign a non-negative weight to each site. Every site knows the weight of all sites in the network. A *quorum* is any set of sites with more than half the total weight of all sites (cf. Section 7.5). Only the one component that has a quorum of sites can process transactions.

Unfortunately, it is possible that *no* component has a quorum. This can occur if the network splits into more than two components, or if sites fail in the component that would have contained a quorum. If this happens, then no component can process transactions. The DBS fails totally.

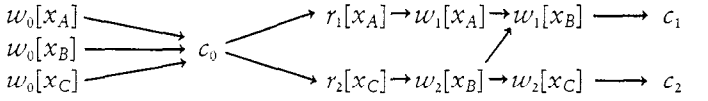
One should assign weights to sites based on their relative importance to the enterprise that is using the DBS. A site with higher weight is more likely to be in the quorum component.

We can improve on the basic site quorum rule by allowing certain transactions to execute in components that don't have a quorum of sites. Since a nonreplicated data item can only be accessed in the component that has its one and only copy, there cannot be two transactions that access a nonreplicated data item but execute in different components. Therefore, a transaction can safely read and write nonreplicated data in any component. For the same reason, a transaction can read and write any replicated data item x in a component that has all of the replicated copies of x. Thus, the site quorum rule need only apply to transactions that access a replicated data item x in a component that is missing one or more copies of x.

---

[17]See Exercise 8.19 for an approach that allows certain conflicting transactions to execute in different components.

The major problem with site quorums is that if transactions have inconsistent views of the components, then they can produce incorrect results. For example, given a database containing $\{x_A, x_B, x_C\}$, suppose site A's TM executes T_1 believing that only A and B are accessible, while site C's TM executes T_2 believing that only B and C are accessible, thereby producing H_8.



TM_A and TM_C both believed they had access to a majority of the sites. But due to a communication anomaly, A and C were each able to communicate with B without being able to communicate with each other. Alternatively, H_8 might have occurred due to the following sequence of events.

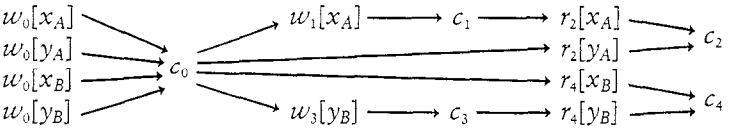
1. A disconnects from B and C.
2. TM_C executes T_2.
3. B disconnects from C and connects to A.
4. TM_A executes T_1.

We'll look at two solutions to this problem: the quorum consensus algorithm, which requires each transaction to access a quorum of *copies* of data items on which it operates (Section 8.9), and the virtual partition algorithm, in which each site synchronizes its view of its component with other sites in its component (Section 8.10). The quorum consensus algorithm has less overhead to handle failures and recoveries than the virtual partition algorithm, but has more overhead to process transactions during periods in which no failures or recoveries take place. Thus, the former is more suitable for environments where failures and recoveries are frequent, and the latter for those where they are relatively rare.

Queries

Since queries (i.e., read-only transactions) do not modify the database, it would seem to be safe to execute them in all components. After all, since each component uses a correct concurrency control algorithm, each query will read a database state produced by a serializable execution of the transactions in the same component. Curiously, however, the execution across two partitions may not be 1SR.

For example, consider the following history.



Sites A and B partition into different components after T_0 executes. In H_9, we allowed queries T_2 and T_4 to read x and y in both partitions. H_9 produces a consistent database state, in the sense that update transactions are 1SR. Moreover, each query reads a consistent database state. Yet there is no serial one-copy history equivalent to H_9, so H_9 is not 1SR.

Whether to allow queries to read all copies in all components is more of a policy issue than a technical one. For most applications the level of data consistency provided by this technique is probably satisfactory. Although anomalies like H_9 are disturbing, the increased data availability to queries will often be the overriding consideration.

Application-dependent Techniques

Site quorums, quorum consensus, and virtual partitions all have the disadvantage of possibly restricting access to copies that are not down. While this restriction is needed to attain one-copy serializability, it does have a cost in lost opportunities. That is, the DBS user is unable to execute certain transactions, and may thereby lose the opportunity to perform certain functions in the real world, resulting in lost revenue, unhappy customers, higher inventory costs, etc. One has to balance these opportunity costs against the value of database integrity. If these costs are high enough, some loss of correctness may be tolerable.

It may therefore be advisable to allow transactions to operate on any available data, even at the risk of non-1SR execution during a communications failure. After the failure is repaired, a special recovery process is invoked to fix database inconsistencies. To make this approach practical for even moderately sized databases, the recovery process needs to be automated.

For example, the recovery process could analyze the logs from sites in different components to find read-write conflicts between transactions in different components. To produce a correct database state, the recovery process could undo and reexecute those transactions as needed. This might entail reexecuting a transaction that committed before the partition (e.g., T_1 or T_2 in H_8), causing it to produce different results than its first execution, a violation of the definition of commitment. Alternatively, the recovery process might only find database inconsistencies by performing validation checks on the data items themselves. Once found, those inconsistencies could be repaired by human intervention, perhaps with the help of a rule-based system.

These approaches may be essential for attaining any appropriately high level of data availability. However, since they allow non-1SR executions to occur, the solutions are necessarily application dependent, and therefore require a highly trained application engineering staff. For users that lack such expertise, general purpose solutions that ensure 1SR executions are usually preferable.

8.9 THE QUORUM CONSENSUS ALGORITHM

In the *quorum consensus* (QC) algorithm, we assign a non-negative weight to each copy x_A of x. We then define a read threshold RT and write threshold WT for x, such that both $2 \cdot \text{WT}$ and $(\text{RT} + \text{WT})$ are greater than the total weight of all copies of x. A *read* (or *write*) *quorum* of x is any set of copies of x with a weight of at least RT (or WT). (Note that the quorums defined here are *copy* quorums, as opposed to the site quorums of the previous section.) The key observation is that each write quorum of x has at least one copy in common with every read quorum and every write quorum of x.

In QC, the TM is responsible for translating Reads and Writes on data items into Reads and Writes on copies. A TM translates each Write(x) into a set of Writes on each copy of some write quorum of x. It translates each Read(x) into a set of Reads on each copy of some read quorum of x, and it returns to the transaction the most up-to-date copy that it read.

To help the TM figure out which copy is most up-to-date, we tag each copy with a *version number*, which is initially 0. When the TM processes Write(x), it determines the maximum version number of any copy it is about to write, adds one to it, and tags all of the versions that it writes with that version number. Clearly, this requires reading all of the copies in the write quorum before writing any of them. This can be done by having Write(x_A) return its version number to the TM. The TM can send the new value and new version number piggybacked on the first round of messages of the atomic commitment protocol.

The version numbers measure how up-to-date each copy is. Each Read of a copy returns its version number along with its data value. The TM always selects a copy in the read quorum with the largest version number (there may be more than one such copy but they will all have the same value).

The purpose of quorums is to ensure that Reads and Writes that access the same data item also access at least one copy of that data item in common. This avoids the problems we saw in histories H_1, H_7, and H_8. Even if some copies are down and are therefore unavailable to Reads and Writes, as long as there are enough copies around to get a read quorum and write quorum, transactions can still continue to execute. Every pair of conflicting operations will always be synchronized by some scheduler, namely, one that controls access to a copy in the intersection of their quorums.

QC works with any correct concurrency control algorithm. As long as the algorithm produces serializable executions, QC will ensure that the effect is just like an execution on a single copy database. To see why, consider any serial execution equivalent to the actual interleaved execution. If a transaction T_j reads a copy of data item x, it reads it from the last transaction before it, T_i, that wrote any copy of x. This is because T_i wrote a write quorum of x, T_j read a read quorum of x, and every read and write quorum has a nonempty intersection. And since T_i is the last transaction that wrote x before T_j read it, T_i

placed a bigger version number on all of the copies of x it wrote than the version number written by any transaction that preceded it. This ensures that T_j will read the value written by T_i, and not by some earlier transaction.

A nice feature of QC is that recoveries of copies require no special treatment. A copy of x that was down and therefore missed some Writes will not have the largest version number in any write quorum of which it is a member. Thus, after it recovers, it will not be read until it has been written at least once. That is, transactions will automatically ignore its value until it has been brought up-to-date.

Unfortunately, QC has some not so nice features, too. Except in trivial cases, a transaction must access multiple copies of each data item it wants to read. Even if there is a copy of the data item in the DM at the site at which it is executing, the transaction still has to look elsewhere for other copies so it can build a read quorum. In many applications, transactions read more data items than they write. Such applications may not perform well using QC.

One might counter this argument by recommending that each read quorum of x contain only one copy of x. But then there can only be one write quorum for x, one that contains all copies of x. This would lead us to the write-all approach in Section 8.1, which we found was unsatisfactory.

A second problem with QC is that it needs a large number of copies to tolerate a given number of site failures. For example, suppose quorums are all majority sets. Then QC needs three copies to tolerate one failure, five copies to tolerate two failures, and so forth. In particular, two copies are no help at all. With two copies QC can't even tolerate one failure.

A third problem with quorum consensus is that all copies of each data item must be known in advance. A known copy of x can recover, but a new copy of x cannot be created because it could alter the definition of x's quorums. In principle, one can change the weights of the sites (and thereby the definition of quorums) while the DBS is running, but this requires special synchronization (see Exercise 8.20).

We will see other approaches to replicated data concurrency control that circumvent QC's weaknesses. Before moving on, though, let's define the histories that QC produces and prove that they are all 1SR.

*Correctness Proof for Quorum Consensus

A *QC history* is an RD history that models an execution of the QC algorithm. Every QC history H has the following properties.

QC_1: If T_i writes x, then H contains $w_i[x_{A_1}], \ldots, w_i[x_{A_n}]$ for some write quorum $wq(x) = \{x_{A_1}, \ldots, x_{A_n}\}$ of x.

QC_1 is simply the rule for using write quorums.

Let $last_i(x) = \{T_j \mid$ for some $x_A \in wq(x)$, $w_j[x_A]$ is the last Write on x_A that precedes $w_i[x_A]$ in $H\}$. Notice that $last_i(x) = \{\}$ if T_i is the first transaction in

H to write x. Define T_i's version number for x to be $\mathrm{VN}_i(x) = 1 + \max\{\mathrm{VN}_j(x) \mid T_j \in \mathrm{last}_i(x)\}$, where $\max\{\} = 0$. Intuitively, $\mathrm{VN}_i(x)$ is the version number that T_i assigns to the copies of x that it writes.

We have to distinguish between the Reads that a transaction applies to the database, and the real Read that is actually selected from among those Reads, that is, the one with the largest version number. We will use $rr_i[x]$, for "real Read," to represent the Read that is selected.

QC_2: If T_j reads x, then H contains $r_j[x_{A_1}], \ldots, r_j[x_{A_n}]$ for some read quorum $rq(x) = \{x_{A_1}, \ldots, x_{A_n}\}$ of x. For each x_B in $rq(x)$, let $\mathrm{VN}(x_B) = \mathrm{VN}_i(x)$ where $w_i[x_B]$ is the last Write on x_B that precedes $r_j[x_B]$ in H. Then H contains $rr_j[x_{A_k}]$ for some x_{A_k} in $rq(x)$ where $\mathrm{VN}(x_{A_k}) = \max\{\mathrm{VN}(x_B) \mid x_B \in rq(x)\}$. Moreover, $r_j[x_A] < rr_j[x_{A_k}]$ for each x_A in $rq(x)$.

QC_2 says that each transaction that reads x reads a read quorum of x and selects from that read quorum a copy with a maximal version number. Since the TM cannot determine the real Read until it knows the version numbers of all copies in $rq(x)$, all Reads of those copies must precede $rr_j[x_{A_k}]$.

QC_3: Every $r_j[x_A]$ follows at least one $w_i[x_A]$, $i \neq j$.

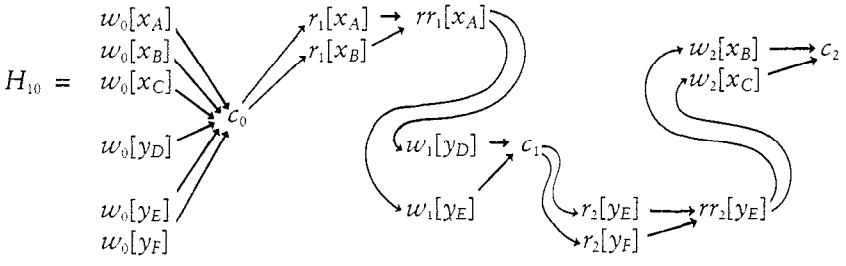
QC_4: $\mathrm{SG}(H)$ is acyclic.

QC_3 requires that each copy be initialized before it can be read. QC_4 says that the schedulers use a correct concurrency control algorithm.

For example, consider a database with data items x and y, with copies x_A, x_B, x_C, y_D, y_E, and y_F. Let the read and write quorums be all majority sets. Consider transactions $\{T_0, T_1, T_2\}$:

$$T_0 = w_0[x] \rightarrow c_0 \leftarrow w_0[y] \qquad T_1 = r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_2 = r_2[y] \rightarrow w_2[x] \rightarrow c_2$$

A possible QC history over these transactions is



In H_{10}, $\mathrm{last}_0(x) = \mathrm{last}_0(y) = \{\}$, hence $\mathrm{VN}_0(x) = \mathrm{VN}_0(y) = 0$; $\mathrm{last}_1(y) = \mathrm{last}_2(x) = \{T_0\}$, hence $\mathrm{VN}_1(y) = \mathrm{VN}_2(x) = 1$. The copies of x that T_1 reads were both written by T_0, hence have identical VNs, and T_1 may use the value of either. The copies of y that T_2 reads were written by different transactions, hence have different VNs; T_2 uses the value of copy y_E, which has the larger

VN, as required. The inclusion of both $r_1[x_A]$ and $rr_1[x_A]$ does not signify that T_1 reads x_A twice; it merely means that x_A is determined to be the copy to be used by T_1 as the value of x.

We will prove that every QC history H is 1SR by proving that SG(H) is an RDSG of H. Since SG(H) is acyclic by QC_4, H is 1SR by Theorem 8.4.

Lemma 8.8: Let H be a QC history. SG(H) induces a write order for H.

Proof: Let T_i and T_k write x. Since all write quorums of a data item intersect, there exists a copy x_A that T_i and T_k both write. These Writes on x_A conflict, so SG(H) must have an edge connecting T_i and T_k. Thus, $T_i \rightarrow T_k$ or $T_k \rightarrow T_i$, so SG(H) induces a write order. □

Lemma 8.9: Let H be a QC history. SG(H) induces a read order for H.

Proof: Suppose T_j reads-x-from T_i, T_k writes x ($i \neq k$ and $j \neq k$), and $T_i \rightarrow T_k$. We need to prove $T_j \rightarrow T_k$. By QC_2, T_j reads a read quorum of x; by QC_1, T_k writes a write quorum of x. Since any two such quorums must intersect, there is a copy of x, say x_B, such that $r_j[x_B]$ and $w_k[x_B]$ are in H. Since these are conflicting operations, either $r_j[x_B] < w_k[x_B]$ or $w_k[x_B] < r_j[x_B]$. In the former case, $T_j \rightarrow T_k$ is in SG(H) and we are done. We now prove that the latter case leads to contradiction.

Let $w_h[x_B]$ be the last Write on x_B preceding $r_j[x_B]$ (possibly $h = k$). Since T_j reads-x-from T_i, by QC_2 $VN_h(x) \leq VN_i(x)$.

We claim that $VN_k(x) \leq VN_h(x)$. If $h = k$, $VN_k(x) = VN_h(x)$. If $h \neq k$, let $w_k[x_B] < w_{h_n}[x_B] < \ldots < w_{h_1}[x_B] < w_h[x_B]$ be the sequence of Writes on x_B between $w_k[x_B]$ and $w_h[x_B]$. For each pair of adjacent Writes in the sequence, say $w_f[x_B] < w_g[x_B]$, by the definition of VN we have $VN_f(x) < VN_g(x)$. So in this case $VN_k(x) < VN_h(x)$. Therefore, $VN_k(x) \leq VN_h(x)$ as claimed. Since $VN_h(x) \leq VN_i(x)$, we also have $VN_k(x) \leq VN_i(x)$.

Since T_i and T_k write x, by QC_1 they write some copy x_C in common. Since $T_i \rightarrow T_k$ and SG(H) is acyclic (by QC_4), it must be that $w_i[x_C] < w_k[x_C]$. Applying the argument of the previous paragraph, we get $VN_i(x) < VN_k(x)$. But this contradicts $VN_k(x) \leq VN_i(x)$. □

Theorem 8.10: The quorum consensus algorithm is correct.

Proof: It is enough to show that any history H that satisfies $QC_1 - QC_4$ is 1SR. By Lemmas 8.8 and 8.9, SG(H) is an RDSG of H. By QC_4, SG(H) is acyclic. By Theorem 8.4, then, H is 1SR. □

The Missing Writes Algorithm

Quorum consensus pays for its resiliency to communications failures by increasing the cost of Reads and by increasing the required degree of replica-

tion. These costs are high if communications failures are infrequent. It would be preferable if the cost could be reduced during reliable periods of operation, possibly at the expense of higher overhead during unreliable periods.

The *missing writes* algorithm is one approach that exploits this trade-off. During a reliable period, the DBS processes Read(x) by reading any copy of x and Write(x) by writing all copies of x. When a failure occurs, the DBS resorts to QC. After the failure is repaired, it returns to reading any copy and writing all copies. Thus, it only pays the cost of QC during periods in which there is a site or communications failure.

Each transaction executes in one of two modes: *normal* mode, in which it reads any copy and writes all copies, or *failure* mode, in which it uses QC. A transaction must use failure mode if it is "aware of missing writes." Otherwise, it can use normal mode.

A transaction is aware of missing writes (MWs) if it knows that a copy x_A does not contain updates that have been applied to other copies of x. For example, if a transaction sends a Write to x_A but receives no acknowledgment, then it becomes aware of MWs. More precisely, transaction T_i *is aware of an MW for copy* x_A in some execution if either T_i writes x but is unable to write x_A, or some transaction T_j is aware of an MW for x_A and there is a path from T_j to T_i in the SG of that execution.

Suppose T_i is aware of an MW for x_A. Then the failure of x_A must precede T_i. But if T_i reads x_A, then T_i precedes the failure of x_A. Since T_i has an inconsistent view of the failure of x_A, it risks producing a non-1SR execution. Consequently, if T_i ever becomes aware of an MW for a copy it read, it must abort.

If T_i is aware of an MW for x_A, it still can read x. But by the preceding argument, it must read a copy of x that isn't missing any Writes. To ensure this, the DBS uses QC. That is, if a transaction that reads (or writes) x is aware of MWs, then it must read (or write) a read (or write) quorum of x.

If T_i begins running in normal mode and becomes aware of an MW as it runs, the DBS can abort T_i and reexecute it in failure mode. Alternatively, it can try to upgrade to failure mode on the fly. That is, for each x that T_i read, the DBS accesses a read quorum R and checks that the value that T_i read is at least as up-to-date as all copies in R. For each x that T_i wrote, the DBS checks that T_i wrote a write quorum of x (see Exercise 8.21).

To implement the algorithm, we need a mechanism whereby a transaction becomes aware of MWs. If a transaction T_i times out on an acknowledgment to one of its Writes, then it immediately becomes aware of an MW. But what if T_j is aware of an MW and $T_j \rightarrow T_i$? The definition of MW awareness requires that T_i become aware of the MWs that T_j is aware of. To do this, T_j should attach a list L of the MWs it is aware of to each copy y_B that it accesses. It tags L to indicate whether it read or (possibly read and) wrote y_B. When T_i accesses y_B in a mode that conflicts with L's tag, it becomes aware of those MWs. The

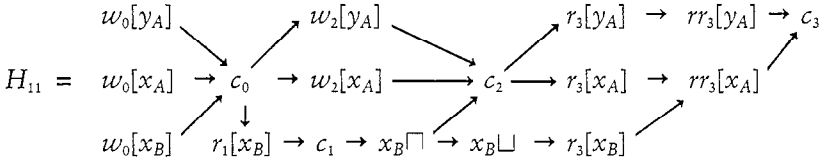
DM should acknowledge T_i's access to y_B by returning a copy of L. T_i can now propagate L, along with other such lists it received, to all of the copies that *it* accesses. This way a transaction T_j propagates MWs that it's aware of to all transactions that follow T_j in the SG, as required by the definition of MW awareness.

After recovering from failure, the DBS at site A has two jobs to do. First, it must bring each newly recovered copy x_A up-to-date. This is easy to do with a copier transaction. The copier simply reads a quorum of copies of x, and writes into all of those copies the most up-to-date value that it read. Version numbers can be used to determine this value, as before.

Second, after a copy x_A has been brought up-to-date, the DBS should delete x_A from the lists of MWs on all copies, so that eventually transactions that access those copies no longer incur the overhead of QC. This entails sending a message to all sites, invalidating entries for x_A on their lists of MWs. The problem is that while these messages are being processed, x_A may fail again and some Write $w_i[x_A]$ may not be applied. Since this MW occurred *after* the recovery of x_A that caused MW entries for x_A to be invalidated, this MW should not itself be invalidated. However, if the new MW is added to an MW list before the old x_A entry in that list was removed, it risks being removed too. To avoid this error, entries in MW lists should contain version numbers. We leave the algorithm for properly interpreting these version numbers as an exercise (see Exercise 8.22).

We can prove the correctness of the MW algorithm using RDSGs, much as we did for QC. As in QC, the SG in MW induces a write order. However, it does not necessarily induce a read order, which makes the proof more complex than for QC.

For example, suppose we have three copies $\{y_A, x_A, x_B\}$, with weight(y_A) = 1, weight(x_A) = 2, and weight(x_B) = 1. The read and write thresholds are: RT(x) = WT(x) = 2, RT(y) = WT(y) = 1. Consider the following history.



In H_{11}, T_0 and T_1 are not aware of MWs and therefore run in normal mode. T_2 is aware of its MW on x_B, and so runs in failure mode. Although x_B recovers before T_3 begins, T_3 is still aware of the MW on x_B and therefore runs in failure mode.

T_1 reads-x-from T_0, T_2 writes x, and $T_0 \rightarrow T_2$ is in SG(H_{11}). To induce a read order, we need $T_1 \rightarrow T_2$ in RDSG(H_{11}). But $T_1 \rightarrow T_2$ is not in SG(H_{11}), so

we need to add it to the SG to make it an RDSG. It turns out that in all such cases where the SG of a history generated by the MW algorithm is missing an edge that is needed to create a read order, we can simply add the necessary edge to the RDSG without creating a cycle. This is the main step in proving every such history has an acyclic RDSG and therefore is 1SR (see Exercise 8.23).

8.10 THE VIRTUAL PARTITION ALGORITHM

As we have seen, the major drawback of the quorum consensus algorithm is that it requires access to multiple, remote copies of x in order to process a Read(x), even when a copy of x is available at the site where the Read is issued. This defeats one of the motivations for data replication, namely, to make data available "near" the place where it is needed. The missing writes algorithm mitigates this problem by using quorum consensus only when site or communication failures exist. The *virtual partition (VP) algorithm*, studied in this section, is designed so that a transaction *never* has to access more than one copy to read a data item. Thus, the closest copy available to a transaction can always be used for reading.

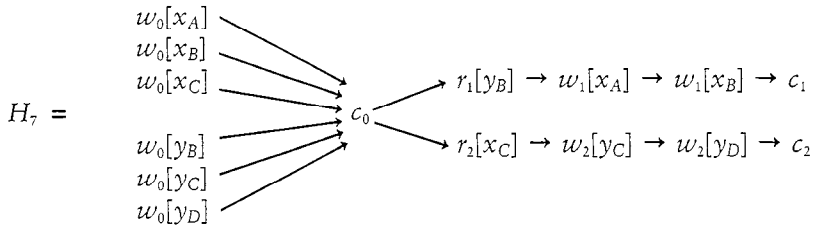
As in quorum consensus, each copy of x has a non-negative weight, and each data item has read and write thresholds (RT and WT, respectively) with the properties described at the beginning of Section 8.9. Read and write quorums are defined as before. However, these serve a different purpose in VP than in QC, as we'll see presently.

The basic idea of VP is for each site A to maintain a *view*, consisting of the sites with which A "believes" it can communicate. We denote this set by $v(A)$. As usual, each transaction T has a home site, $home(T)$, where it was initiated. The view of a transaction, $v(T)$, is the view of its home site, $v(home(T))$, at the time T starts. As we will see shortly, if $v(home(T))$ changes before T commits, then T will be aborted.

A transaction T executes as if the network consisted just of the sites in its view. However, for the DBS to process a Read(x) (or Write(x)) of T, $v(T)$ *must* contain a read (or write) quorum for x. If that is not the case, the DBS must abort T. Note that the DBS can determine if it can process Read(x) or Write(x) from information available at $home(T)$. It need not actually try to access a read or write quorum of x's copies.

Within the view in which T executes, the DBS uses the write-all approach. That is, it translates Write(x) into Writes on all copies of x in $v(T)$; it translates Read(x) into a Read of *any* copy of x in $v(T)$.

A good illustration of how this idea works is provided if we examine how VP avoids H_7 — our archetype of a problematic history in the presence of communication failures.



Recall that in this example, the network is partitioned into two components, $\{A, B\}$ and $\{C, D\}$. Suppose $home(T_1) = A$ and $home(T_2) = C$. If each site has a correct view of the sites with which it can communicate, $v(A) = \{A, B\}$ and $v(C) = \{C, D\}$. If the DBS allows T_1 to commit in H_7, $v(A)$ must have a read quorum of y (and a write quorum of x), and therefore $v(C)$ can't have a write quorum of y (or a read quorum of x). Hence the VP algorithm will not allow T_2 to commit.

Views of sites change, as site and communication failures cause the network to be reconfigured. Hence, a transaction T may attempt to communicate with a site whose view is different from T's. Or, T may attempt but fail to communicate with a site in its view. (A transaction communicates with a site to access a copy stored there or send a VOTE-REC during its commitment). These situations indicate that $home(T)$'s view is not accurate. T must abort and $home(T)$'s view must be adjusted, as explained next. Moreover, whenever a site A's view changes, all active transactions whose home is A must abort. They can all then be restarted and try to run within A's new view.

Maintaining the views in a consistent manner is not a simple matter, because site and communication failures occur spontaneously. VP surmounts this difficulty by using a special *View Update transaction*. This is issued by the DBS itself, rather than by users. In spirit it is analogous to the Include and Exclude transactions used by the directory-oriented available copies algorithm to maintain a consistent view among all sites of the copies and directories that are available (cf. Section 8.7).

When a site detects a difference between its present view and the set of sites it can actually communicate with, it initiates a View Update transaction whose purpose is:

1. to adjust the views of all the sites in the new view, and
2. to bring up-to-date all copies of all data items for which there is a read quorum in the new view.

In adjusting the views of the sites in a new view, View Updates must be coordinated to avoid problems like the one exemplified by the following scenario. Consider a network with four sites A, B, C, D such that at some point $v(B) = \{B\}$; $v(D) = \{D\}$, and $v(A) = v(C) = \{A, C\}$. Later, B discovers that it can communicate with A and C (but not with D) and, at the same time,

D discovers that it can communicate with A and C (but not with B).[18] Thus B will initiate a View Update to create view $\{A, B, C\}$ and D a similar one to create view $\{A, C, D\}$. Unless these two activities are properly coordinated, it is possible that A joins the first view and C the second, creating a situation where $v(A) = v(B) = \{A, B, C\}$ and $v(C) = v(D) = \{A, C, D\}$. If such inconsistent views were allowed to form, H_7 *would* be allowed since the views of A and C (home sites of T_1 and T_2, respectively) would both contain read and write quorums for x and y!

To avoid such inconsistencies, VP uses a *view formation protocol*. Associated with each view is a unique *view identifier* (*VID*). When a site A wishes to form a new view, say v, it generates a VID, newVID, greater than its present VID. A then initiates a two phase protocol to establish the new view:

- A sends a JOIN-VIEW message to each site in v and waits for acknowledgments. This message contains newVID.
- Receipt of JOIN-VIEW by site B constitutes an invitation for B to join the new view being created by A. B accepts the invitation provided its present VID is less than the newVID contained in JOIN-VIEW; otherwise, it rejects the invitation. Accepting the invitation means sending a positive acknowledgment; rejecting it means sending a negative acknowledgment (or nothing at all).
- After receiving acknowledgments, A can proceed in one of two ways. It may abort the creation of the new view (either by sending explicit messages to that effect or by not sending any messages at all). In this event, A may attempt to restart the protocol using a greater newVID, hoping it will convince more sites to accept the invitation. Alternatively, A may ignore any rejected invitations and proceed to form a new view consisting only of the set of sites v' ($v' \subseteq v$) that accepted the invitation. It does this by sending a VIEW-FORMED message to each site in v' and attaching the set v' to that message. Any site that receives that message adopts v' as its view and newVID (which it had received in the JOIN-VIEW message) as its present VID. It need not acknowledge the receipt of VIEW-FORMED.

Returning to our example, if this protocol is followed, A and C will make a consistent choice: Both will join either the view initiated by B ($\{A, B, C\}$) or the view initiated by D ($\{A, C, D\}$), depending on which of the two had a greater VID. A third possibility, namely, that neither A nor C joins either new view, would obtain if the present VID of A and C is greater than the VIDs of the views B and D are attempting to create.

To ensure network-wide uniqueness, VIDs are pairs (c, s) where s is a site identifier and c is a counter stored at s and incremented each time s tries to

[18]Recall that timeout failures can give rise to such anomalous situations (cf. Section 7.2).

create a new view. Thus VIDs created by different sites differ in the second component, while VIDs created by the same site differ in the first component. Pairs are compared in the usual way: $(c, s) < (c', s')$ if $c < c'$, or $c = c'$ and $s < s'$.

When a new view is created, all copies of data items for which there is a read quorum in the view must be brought up-to-date. This is because, in the new view, any such copy might be read. The task of updating such copies is also carried out by the View Update transaction. It is done as follows. When a site A creates a new view v, it reads a read quorum of each data item x (for which there is a read quorum in v). It then writes all copies of x in v, using the most recent value it read. Version numbers can be used to determine the most recent value, as in QC. This procedure need not be carried out for copies that A can tell are up-to-date (see Exercise 8.24). The entire process of updating copies can be combined with the view formation protocol (see Exercise 8.25). This whole activity comprises what we called before a View Update transaction. It should be emphasized that the activity of a View Update is carried out like that of an ordinary (distributed) transaction. In particular, Reads and Writes issued in the process of bringing copies of the new view up-to-date are synchronized using the DBS's standard concurrency control mechanism. In the VP algorithm we described, the set of copies of each data item is fixed. Adding new copies of a data item x alters the quorum of x. Therefore, as in QC, it requires special synchronization (see Exercise 8.26).

VP guarantees that an execution is 1SR by ensuring that all transactions see failures and recoveries in the same order. Informally, we can argue this as follows.

1. Each transaction executes entirely within a single view.
2. For any two transactions T_i and T_j, if $T_i \rightarrow T_j$, then T_i executed in a view whose VID is less than or equal to T_j's.
3. From (2), there is a serial history H_s that is equivalent to the one that actually occurred in which transactions in the same view are grouped into contiguous subsequences, ordered by their VIDs.

Within a VP, all transactions have the same view of which copies are functioning and which are down. So if we imagine site failures and recoveries to occur at the beginning of each segment in H_s, then all transactions have a consistent view of those failures and recoveries. And this, as we know, implies that the execution is 1SR (see Exercise 8.27).

BIBLIOGRAPHIC NOTES

Since the earliest work on distributed databases, replication has been regarded as an important feature [Rothnie, Goodman 77] and [Shapiro, Millstein 77b]. Some early algorithms include primary site [Alsberg, Day 76] [Alsberg et al. 76], majority consensus [Thomas 79], primary copy [Stonebraker 79b], and a TO-based write-all-available algorithm in SDD-1 [Bernstein, Shipman, Rothnie 80] and [Rothnie et al. 80].

The concept of one copy serializability was introduced in [Attar, Bernstein, Goodman 84]. The theory of 1SR histories was developed in [Bernstein, Goodman 86a] and [Bernstein, Goodman 86b].

Available copies algorithms are described in [Bernstein, Goodman 84], [Chan, Skeen 86], and [Goodman et al. 83]. Weak consistency for queries is discussed in [Garcia-Molina, Wiederhold 82]. Majority consensus was generalized to quorum consensus in [Gifford 79]. Other majority and quorum based algorithms are presented in [Breitwieser, Leszak 82], [Herlihy 86], and [Sequin, Sargeant, Wilnes 79]. The missing writes algorithm was introduced in [Eager 81] and [Eager, Sevcik 83]. The virtual partition algorithm was introduced in [El Abbadi, Skeen, Cristian 85] and [El Abbadi, Toueg 86].

[Davidson, Garcia-Molina, Skeen 85] gives a survey of approaches to replication, including some methods not described in this chapter: [Skeen, Wright 84], which describes a method for analyzing transaction conflicts to determine when a partition cannot lead to a non-1SR execution (see Exercise 8.19); [Davidson 84], which shows how to analyze logs to determine which transactions must be undone after a network partition is repaired; and [Blaustein et al. 83] and [Garcia-Molina et al. 83b], which describe methods for recovering an inconsistent database after a partition is repaired by exploiting semantics of transactions.

EXERCISES

8.1 The primary copy approach to the distribution of replicated Writes is useful for avoiding deadlocks in 2PL. Explain why it eliminates deadlocks resulting from write-write conflicts. Is it helpful in reducing other types of deadlock, too?

8.2 Suppose we use 2PL with the primary copy and write-all approaches to the distribution of Writes. Assume there are no failures or recoveries. Must updaters set write locks on non-primary copies? How would you change your answer if queries are allowed to read non-primary copies?

8.3 Consider an algorithm for creating new copies that satisfies the following three conditions:

a. it produces serializable executions (its RD histories have acyclic SGs),
b. no transaction can read a copy until it has been written into at least once, and
c. once a transaction T_i has written into a new copy x_A, all transactions that write into x and come after T_i in the SG also write into x_A.

Prove or disprove that such an algorithm produces only 1SR executions.

8.4 Consider the following algorithm for creating a new copy x_A of x. The DBS uses primary copy for distributing Writes. The DBS produces serializable executions (its RD histories have acyclic SGs). The scheduler that controls x_A does not allow x_A to be read until it has been written into

at least once. Once the primary copy's DBS has sent a Write on x to x_A, it will send to x_A all subsequent Writes on x that it receives. Assuming that copies never fail, prove that this algorithm produces 1SR histories.

8.5 Suppose a site A fails for a short period. Most data items at A that are replicated elsewhere were probably not updated by any other site while A was down. If other sites can determine the earliest time τ that A might have failed, then using τ they can examine their logs to produce a list of data items that were updated while A was down. These are the only data items at A that must be reinitialized when A recovers. Propose a protocol that sites can use to calculate τ.

8.6 One way to avoid copiers on all data items is for a site B to step in as A's *spooler* after A fails. B spools to a log the Writes destined for A (which A can't process, since it's down). When A recovers, it uses a Restart procedure based on logging to process the log at B (cf. Section 6.4). What properties must B's spooler and A's Restart procedure satisfy for this approach to work correctly? Given those properties, how should A respond to Reads while it is processing B's log?

8.7 A serial RD history H is called *1-serial* if, for all i, j, and x, if T_j reads-x-from T_i $(i \neq j)$, then T_i is the last transaction preceding T_j that writes into any copy of x. Prove that if an RD history is 1-serial, then it is 1SR.

8.8* Let H be an RD history over T. Let $G(H)$ be a directed graph whose nodes are in one-to-one correspondence with the transactions in T. $G(H)$ *contains a reads-from order* if, whenever T_j reads-x-from T_i,

a. if $(i \neq j)$, then $T_i \lll T_j$, and
b. if for some copy x_A, $w_i[x_A] < r_i[x_A]$, $w_k[x_A] \in H$ and $T_k \lll T_i$ then $w_k[x_A] < w_i[x_A]$.

$G(H)$ is a *weak replicated data serialization graph* (*weak RDSG*) for H if it contains a write order, a reads-from order, and a read order. Prove that H has the same reads-from relationships as a serial 1C history over T if *and only if* H has an acyclic weak RDSG. Give an example of an RD history that does not have an acyclic RDSG but *does* have an acyclic weak RDSG.

Since a weak RDSG need not contain $SG(H)$, Theorem 8.3 does not apply. Thus, we are not justified in dropping final writes from a proof that H is equivalent to a serial 1C history. Prove or disprove that H is 1SR iff it has an acyclic weak RDSG.

8.9* In the definition of FRSGs, we assumed that each copy is created once, fails once, and then never recovers. Suppose we drop the assumption that failed copies never recover. To cope with multiple failures and recoveries of a copy, we extend an FRSG to be allowed to include more than one pair of nodes $\{\text{create}_i[x_A], \text{fail}_i[x_A]\}$ for each copy x_A, where the subscripts on create and fail are used to relate matching create/fail pairs. Redefine the edge set of an FRSG to make use of these multiple create/fail pairs. Prove Theorem 8.5 for this new definition of FRSG.

8.10* Suppose we add the following set of edges to an FRSG:

$$E2' = \{T_i \rightarrow \text{fail}[x_A] \mid T_i \text{ writes } x_A\}.$$

An *augmented FRSG* is an FRSG that includes the edges defined by E2′.

a. Give an example history that has an acyclic FRSG, but has no acyclic augmented FRSG.
b. Suppose a concurrency control algorithm only produces histories that have acyclic augmented FRSGs. Explain intuitively what effect this has on the state of a copy after it recovers from a failure.
c. Does the available copies algorithm of Section 8.6 produce histories that have acyclic augmented FRSGs? If not, modify the algorithm so that it does.

8.11 Give an example of a non-1SR execution that would be allowed by the simple available copies algorithm of Section 8.6, if the missing writes validation and the access validation steps were done at the same time (i.e., the validation protocol sent the AVAILABLE and UNAVAILABLE messages at the same time).

8.12* Give a proof of correctness for the simple available copies algorithm (Section 8.6) based on Theorem 8.4. That is, list conditions satisfied by histories produced by that algorithm, justify why they must be satisfied, and prove that any history that satisfies them must have an acyclic RDSG.

8.13 When all copies of a data item fail, we say the item has suffered a total failure. In available copies algorithms special care must be taken in recovering from a total failure of a data item: The copier transaction that brings copies up-to-date must read the value of the copy that failed last. Describe methods for recovering from such total failures, for available copies algorithms.

8.14 How do the quorum consensus, missing writes, and virtual partition algorithms handle total failures of data items?

8.15 In the directory-oriented available copies algorithm, a transaction can perform access validation by checking that all directories from which it read contain the copies it accessed. Why is this so? That is, why is it not necessary to access the copies themselves? Also, if the scheduler uses Strict 2PL, show that access validation requires merely to check that the directories read by the transaction are still available (their contents need not be considered).

8.16* Formalize the correctness argument given in Section 8.7 to derive a proof that the directory-oriented available copies algorithm produces only 1SR executions.

8.17* Give a correctness proof for the directory-oriented available copies algorithm, based on Theorem 8.4. That is, prove that any history representing an execution produced by that algorithm must have an acyclic RDSG.

8.18 Describe, in some detail, an algorithm that allows the creation and destruction of directory copies, using a (replicated) master directory.

8.19* Suppose each site is assigned a set of classes and can only execute transactions that are in its classes. (See Section 4.2 for a description of classes.) Assume that the readset of each class contains its writeset.

Suppose the network has partitioned. Define a directed graph G each of whose nodes represents the occurrence of a given class in a given partition. For each pair of nodes C_i, C_j where readset(C_i) $\cap$ writeset(C_j) $\neq \{\}$, G contains the edge $C_i \rightarrow C_j$ and, if C_i and C_j are in the same partition, $C_j \rightarrow C_i$. A *multipartition cycle* in G is a cycle that contains two or more nodes whose sites are in different partitions.

Assume that the partitioned network experiences no site or communication failures or recoveries. Suppose the DBS in each partition uses the write-all approach with respect to the copies in its partition. Prove that if G has no multipartition cycles, and SG(H) is acyclic for RD history H, then H is 1SR.

8.20 Describe a method whereby the weights of copies can be changed dynamically in the quorum consensus algorithm.

8.21 In the missing writes algorithm a transaction begins by using the write-all approach. If it should ever become aware of missing writes, it can abort and rerun using quorum consensus. Alternatively, it can try to switch to quorum consensus "on the fly." Describe, in some detail, the latter option. (Hint: Consider doing some extra work during the atomic commitment protocol.)

8.22 Describe, in some detail, a technique whereby missing writes to a copy are "forgotten" when all such Writes have actually been applied to the copy. Make sure your algorithm can handle correctly the situation where a copy misses (again) some updates while (old) missing updates to that copy are being "forgotten."

8.23* Give a precise proof of correctness for the missing writes algorithm by giving a list of properties satisfied by histories that represent executions of that algorithm and then proving that any history that satisfies these conditions must have an acyclic RDSG.

8.24 Consider the formation of a new view in the virtual partition algorithm. Which copies of items for which there is a read quorum in the new view are guaranteed to be up-to-date?

8.25 Describe, in some detail, a View Update transaction, by integrating the view formation protocol and the copy update process. Take into account your answer to the previous exercise to optimize the latter. Explore the round and message complexity of a View Update transaction.

8.26 Describe a method whereby the weights of copies can be changed dynamically in the virtual partition algorithm.

8.27* Give a formal proof of correctness for the virtual partition algorithm.