Consistency in Partitioned Networks

Susan B. Davidson

Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania 19104

Hector Garcia-Molina

Department of Computer Science, Princeton University, Princeton, New Jersey 08540

Dale Skeen

IBM Almaden Research Center, San Jose, California 95120

Recently, several strategies have been proposed for transaction processing in partitioned distributed database systems with replicated data. These strategies are surveyed in light of the competing goals of maintaining correctness and achieving high availability. Extensions and combinations are then discussed, and guidelines are presented for selecting strategies for particular applications.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*reliability, availability, and serviceability*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Consistency, network partitioning, serializability

INTRODUCTION

In a distributed database system, data are often replicated to improve performance and availability. By storing copies of shared data on processors where they are frequently accessed, the need for expensive, remote read accesses is decreased. By storing copies of critical data on processors with independent failure modes, the probability that at least one copy of the data will be accessible increases. In theory, data replication makes it possible to provide arbitrarily high data availability.

In practice, realizing the benefits of data replication is difficult since the *correctness* of data must be maintained. One important aspect of correctness with replicated data is *mutual consistency*: All copies of the same logical data item must agree on exactly one "current value" for the data item. Furthermore, this value should "make sense" in terms of the transactions executed on copies of the data item. When communication fails between sites containing copies of the same logical data item, mutual consistency between copies becomes complicated to ensure. The most disruptive of these communication failures are *partition failures*, which fragment the network into isolated subnetworks called *partitions*. Unless partition failures are detected and recognized by all affected processors, independent and uncoordinated updates may be applied to

CONTENTS

different copies of the data, thereby compromising the correctness of data. Consider, for example, an airline reservation system implemented by a distributed database that splits into two partitions when the communication network fails. If, at the time of the failure, all the nodes have one seat remaining for PAN AM 537, reservations could be made in both partitions. This would violate correctness: Who should get the last seat? There should not be more seats reserved for a flight than physically exist on the plane. (Some airlines do not implement this constraint and allow overbookings.)

The design of a replicated data management algorithm tolerating partition failures is a notoriously hard problem. Typically, the cause or extent of a partition failure cannot be discerned by the processors themselves. At best, a processor may be able to identify the other processors in its partition; but, for the processors outside of its partition, it will not be able to distinguish between the case in which those processors are simply isolated from it and the case in which those processors are down. In addition, slow responses from certain processors can cause the network to appear partitioned even when it is not, further complicating the design of a fault-tolerant algorithm.

As far back as 1977, partitioned operation was identified as one of the important and challenging open issues in distributed data management [Rothnie and Goodman 1977]. Since then our understanding of the problem has increased dramatically, and a number of diverse solutions have been proposed. In this paper, we survey several of the more general solutions, and discuss current research trends in this still young and active research area.

Although our discussion is couched within a database context, most results have more general applications. In fact, the only essential notion in many cases is that of a transaction. Hence these strategies are immediately applicable to mail systems, calendar systems, object-oriented systems, and other applications using transactions as their underlying model of processing.

The remaining sections of the survey are organized as follows. Section 1 is a discussion of the principal consideration in designing a processing strategy for a partitioned system: the trade-off between correctness and availability. In Section 2 the notion of correctness in a replicated database system is discussed, and a taxonomy of partition-processing algorithms is introduced. Sections 3 and 4 are surveys of the current solutions for transaction processing while the system is partitioned, and extensions and combinations are suggested. A somewhat different problem is discussed in Section 5: how to complete transactions that are in progress at the time of a partition failure. Guidelines for selecting a partition strategy are presented in Section 6, along with suggestions for future research.

1. CORRECTNESS VERSUS AVAILABILITY

When designing a system that will operate when it is partitioned, the competing goals of availability (the system's normal function should be disrupted as little as possible) and correctness (data must be correct when recovery is complete) must somehow

be met. These goals are not independent; hence trade-offs are involved.

Correctness can be achieved simply by suspending operation in all but one of the partition groups and forwarding updates at recovery; but this severely compromises availability. In applications in which partitions either occur frequently or occur when access to the data is imperative, this solution is not acceptable. For example, in the airline reservation system it may be too expensive to have a high-connectivity network, and partitions may occasionally occur. Many transactions are executed each second (TWA's centralized reservations system estimates 170 transactions per second at peak time [Gifford and Spector 1984]), and each transaction that is not executed may represent the loss of a customer. In a military command and control application, a partition can occur because of an enemy attack, and it is precisely at this time that we do not want transaction processing halted.

On the other hand, availability can be achieved simply by allowing all nodes to process transactions "as usual" (note that transactions can only execute if the data that they reference are accessible). Correctness may now be compromised, however. Transactions may produce "incorrect" results (e.g., reserving more seats than physically available), and the databases in each group may diverge. In some applications, such "incorrect" results may be acceptable in light of the higher availability achieved. When partitions are reconnected, the problems may be corrected by executing transactions missed by a partition, and by choosing certain transactions to "undo." If the chosen transactions have had no real-world effects, they can be undone by using standard database recovery methods. If, on the other hand, they have had real-world effects, then appropriate *compensating transactions* must be run, transactions that not only restore the values of the changed database items but also issue real-world actions to nullify the effects of the chosen transactions (e.g., by canceling certain reservations and sending messages to affected users). Alternatively, *correcting transactions* can be run, transforming the database from an incorrect state to a correct state without undoing the effects of any previous transactions. For instance, in a banking application, the correcting transaction for overdrawing a checking account during a partitioning would apply an overdraft charge. Of course, in some applications incorrect results are either unacceptable or incorrectable. For example, it may not be possible to undo or correct a transaction that effectively hands $1,000,000 to a customer.

Since it is clearly impossible to satisfy both goals simultaneously, one or both must be relaxed to some extent, depending on the application's requirements. Relaxing availability is fairly straightforward; you simply disallow certain transactions at certain sites. Relaxing correctness, on the other hand, usually requires extensive knowledge about what the information in the database represents, how applications manipulate the information, and how much undoing/correcting/compensating inconsistencies will cost. The first step in choosing a partition-processing strategy is to determine which is more important, correctness or availability; the second step is to try to understand the trade-offs between the two properties for the database at hand.

2. THE NOTION OF CORRECTNESS

What does correct processing mean in a database system? Informally, a database is correct if it correctly describes the external objects and processes that it is intended to model. In theory, such a vague notion of correctness could be formalized by a set of static constraints on objects and their attributes, and a set of dynamic constraints on how objects can interact and evolve. In practice, a complete specification of the constraints governing even a small database is impractical (besides, even if it were practical, enforcing the constraints would not be). Consequently, database systems use a less ambitious, very general notion of correctness based on the order of transaction execution and on a small set of static data constraints known as *integrity constraints*.

In this section, we examine the notion of correctness, beginning informally with examples illustrating incorrect behavior, followed by a more formal definition of correctness in the traditional database system. When referring to the state of the database, we use the terms "correct" and "consistent" interchangeably.

2.1 Anomalies

Consider a banking database that contains a checking account and a savings account for a certain customer, with a copy of each account stored at both site A and site B. Suppose that a communication failure isolates the two sites. Figure 1 shows the result of executing a checking withdrawal at A (for \$100) and two checking withdrawals at B (totaling \$100). Although the resulting copies of the checking account contain the same value, we know intuitively that the actions of the system are incorrect: The account owner extracted \$200 from a checking account containing only \$100. The anomaly is caused by conflicting write operations issued in parallel by transactions executing in different partitions.

An interesting aspect of this example is that in the resulting database all copies are mutually consistent;[1] that is, all copies of a data item contain the same value. Thus, although it is commonly used as the correctness criterion for replicated file systems and information databases, such as telephone directories, mutual consistency is not a sufficient condition for correctness in a transaction-oriented database system. It is also not a necessary condition: Consider the example in which A executes the \$100 withdrawal while B does nothing. Although the resulting copies of the checking account contain different values, the resulting database is correct if the system recognizes that the value in A's copy is the most recent one.

A different type of anomaly on the same database is illustrated in Figure 2. This

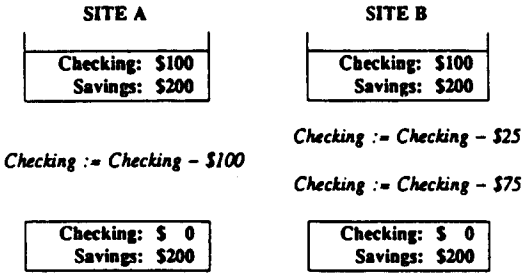

Figure 1. An anomaly resulting from concurrent write operations on the same data item in separate partitions.

figure shows the result of executing a checking withdrawal of \$200 at site A, and a savings withdrawal of \$200 at site B. Here, we assume that the semantics of the checking withdrawal allow the account to be overdrawn as long as the overdraft is covered by funds in the savings account (i.e., $checking + savings \geq 0$). The semantics of the savings withdrawal are similar. In the execution illustrated, however, these semantics are violated: \$400 is withdrawn, whereas the accounts together contain only \$300. The anomaly was not caused by conflicting writes (none existed since the transactions updated different accounts), but instead as a result of the fact that accounts are allowed to be read in one partition and updated in another.

Concurrent reads and writes in different partitions are not the only sources of inconsistencies in a partitioned system; more will be identified shortly. Nor do they always cause inconsistencies: For example, if the savings withdrawal in Figure 2 is changed to a deposit, the intended semantics of the database would not be violated. However, the above are typical anomalies that can occur if conflicting transactions are executed in different partitions.

2.2 Database Model

A database is a set of *logical data items* that support the basic operations *read* and *write*. The granularity of these items is not important: They could be records, files, relations, etc. The *state* of the database is an assignment of values to the logical data items. For brevity, logical data items are

[1] This is the narrowest interpretation of several uses of the term "mutual consistency" that appear in the literature. Some authors use mutual consistency synonymously with one-copy equivalence (defined in Section 2.2).

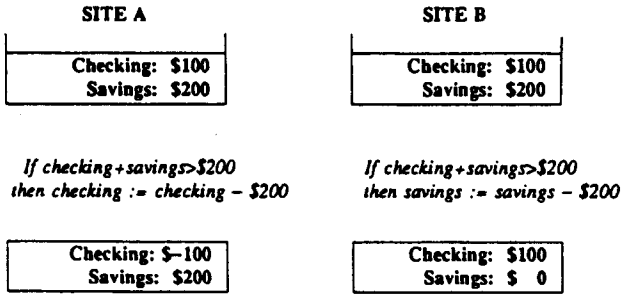

Figure 2. An anomaly resulting from concurrent read and write operations in different partitions.

subsequently called data items or, more simply, items.

A *transaction* is a program that issues read and write operations on the data items. In addition, a transaction may have effects that are external to the database, such as dispensing money or displaying results on a user's terminal. The items read by a transaction constitute its *readset*; the items written constitute its *writeset*. A *read-only transaction* neither issues write requests nor has external effects. Transactions are assumed to be correct. More precisely, *a transaction, when executed alone, transforms an initially correct database state into another correct state* [Traiger et al. 1982].

Transactions interact with one another indirectly by reading and writing the same data items. Two operations on the same item are said to *conflict* if at least one of them is a write. Conflicts are often labeled either *read–write*, *write–read*, or *write–write*, depending on the types of data operations involved and their order of execution [Bernstein and Goodman 1981]. Conflicting operations are significant because their order of execution affects the final database state.

A generally accepted notion of correctness for a database system is that it executes transactions so that they appear to users as indivisible, isolated actions on the database. This property, referred to as *atomic* execution, is achieved by guaranteeing the following properties:

(1) The execution of each transaction is an "all or nothing": Either all of the transaction's writes and external operations are performed or none are performed. (In the former case the transaction is said to be *committed*; in the latter case it is said to be *aborted*.) The property is often referred to as *atomic commitment*.
(2) The execution of several transactions concurrently produces the same database state as some serial execution of the same transactions. The execution is then said to be *serializable*.

The first property is established by the commit and recovery algorithms of the database system; the second is established by the concurrency control algorithm.

Atomic transaction execution (the concurrent execution of transactions is serializable), together with the assumption that transactions are correct (a transaction executed alone transforms an initially correct database state into another correct state), implies by induction that the execution of any set of transactions transforms an initially correct database state into a new, correct state. Although atomic execution is not always necessary to preserve correctness (as we discuss in Section 4), most real database systems implement it as their sole criterion of correctness. This is because atomic execution is simple (it corresponds to users' intuitive model that transactions are processed sequentially) and can be enforced by very general mechanisms that determine the order of conflicting data operations. These mechanisms are independent of both the semantics of the data being stored and the transactions manipulating it.

Some systems allow additional correctness criteria to be expressed in the form of *integrity constraints*. Unlike atomicity, these are semantic constraints. They may range from simple constraints (e.g., the balance of checking accounts must be

nonnegative) to elaborate constraints that relate the values of many data items. In systems enforcing integrity constraints, a transaction is allowed only if its execution is atomic and its results satisfy the integrity constraints. To simplify the discussion, throughout the rest of the paper, we assume that integrity constraints are checked as part of the normal processing of a transaction.

Notice that we have not specified whether we were discussing a centralized or a distributed database system; it has not been necessary to do so since the definitions, the properties of transaction processing, and the correctness criteria are the same in both. Of course, the algorithms for achieving correct transaction processing differ markedly between the two types of implementations.

In a *replicated database*, the value of each logical item x is stored in one or more *physical data items*, which are referred to as the *copies* of x. Each read and write operation issued by a transaction on some logical data item must be mapped by the database system to corresponding operations on physical copies. To be correct, the mapping must ensure that *the concurrent execution of transactions on replicated data is equivalent to a serial execution on nonreplicated data*, a property known as *one-copy serializability*. The logic that is responsible for performing this mapping is called the *replica control algorithm*.

As a correctness criterion, one-copy serializability is attractive for the same reasons that (normal) serializability is: It is intuitive, and it can be enforced using general-purpose mechanisms that are independent of the semantics of the database and of the transactions executed.

The literature on the model and problems discussed above is extensive. The transaction concept was first introduced by Eswaran et al. [1976]. A single-site recovery algorithm is presented by Gray et al. [1981]. Single-site concurrency control algorithms are too numerous to list, but three influential proposals are two-phase locking [Eswaran et al. 1976], timestamp ordering [Bernstein and Goodman 1980], and optimistic concurrency control [Kung and Robinson 1981]. The seminal paper on serializability theory was written by Papadimitriou [1979]. The enforcement of integrity constraints is discussed by Blaustein [1981]. The article by Gray [1978] contains an in-depth treatment of many issues in the implementation of a database system.

For nonpartitioned distributed database systems, concurrency control algorithms are surveyed by Bernstein and Goodman [1981] and Kohler [1981]. Atomic commitment protocols are discussed by Gray [1978], Hammer and Shipman [1980], and Skeen [1982b]. Replica control algorithms are contained in Gifford [1979], Stonebraker [1979], and Goodman et al. [1983]. A good discussion of the requirements for maintaining one-copy serializability in the presence of failures can be found in Bernstein and Goodman [1983].

2.3 Partitioned Operation

Let us now consider transaction processing in a partitioned network, where the communication connectivity of the system is broken by failures or by anticipated communication shutdowns. To keep the exposition simple, let us assume that the network is "cleanly" partitioned (that is, any two sites in the same partition can communicate and any two sites in different partitions cannot communicate) and that one-copy serializability is the correctness criterion.

When the system is partitioned, each partition must determine which transactions it can execute without violating the correctness criteria. Actually, this can be thought of as two problems: (1) each partition must maintain correctness within the part of the database stored at the sites comprising the partition, and (2) each partition must make sure that its actions do not conflict with the actions of other partitions, so that the database is correct across all partitions.

If we assume that each site in the network is capable of detecting partition failures, then correctness *within* a partition can be maintained by adapting one of the standard replica control algorithms for

nonpartitioned systems. For example, the sites in a partition can implement a write operation on a logical object by writing all copies in the partition. This, along with a standard concurrency control protocol, ensures one-copy serializability in the partition.

The really difficult problem is ensuring one-copy serializability *across* partitions. As illustrated in Figures 1 and 2, the transactions in each partition may be one-copy serializable, but conflicting operations can take place in different partitions. Thus it is not sufficient to run a correct replica control algorithm in each partition to ensure that overall transaction execution is one-copy serializable.

A number of solutions have been proposed for keeping data globally consistent, and most of the remainder of the survey is devoted to discussing these solutions. Many of these solutions are based on the simple observation that a sufficient (but not necessary) condition for correctness is that no two partitions execute conflicting data operations. However, not all partition-processing solutions use one-copy serializability as their correctness criterion, nor do all attempt to maintain correctness across partitions. We discuss these alternatives in Section 2.4.

In theory, a partition-processing strategy is composed of two algorithms: one to ensure correctness across partitions and a replica control algorithm to ensure one-copy behavior. In practice, many strategies are composed of a single algorithm that solves both problems. Most "single" algorithms do not require partitions to be detected and tolerate more than just "clean" network failures. Such algorithms are attractive for their additional fault tolerance. In Sections 3 and 4, we present these "single algorithms," along with "partition control" algorithms. In both, however, we emphasize the partition control aspect.

In addition to solving the problem of global correctness, a partition-processing strategy must solve two problems of a different sort. First, when the partitioning occurs, the database is faced with the problem of atomically committing ongoing transactions. The complication is that the sites executing the transaction may find themselves in different partitions, and thus unable to communicate a decision as to whether to complete the transaction (commit) or to undo it (abort). Note that the problem of atomic commitment in multiple partitions does not arise for a transaction submitted after the partitioning occurs (such a transaction will be executed in only one partition) and that this problem arises in any partitioned database system whether it is replicated or not.

Second, when partitions are reconnected, mutual consistency[2] between copies in different partitions must be reestablished. That is, the updates made to a logical data object in one partition must be propagated to its copies in the other partitions. Conceptually, this problem can be solved in a straightforward manner by extra bookkeeping whenever the system partitions. For example, each update applied in a partition can be logged, and this log can be sent to other partitions upon reconnection. (Such a log may be integrated with the "recovery log" that is already kept by many systems.) In practice, an efficient solution to this problem is likely to be intricate and very dependent on the normal recovery mechanisms employed in the database system. For this reason, we do not discuss it further.

2.4 Classification of Strategies

Partition-processing strategies can be classified along two orthogonal dimensions. The first dimension concerns the trade-off between consistency and availability; the two extremes are *pessimistic* and *optimistic*. The second dimension concerns the type of information used in determining correctness; the two extremes are *syntactic* and *semantic*. Thus a strategy can be loosely classified as either pessimistic-syntactic, optimistic-syntactic, pessimistic-semantic, or optimistic-semantic.

Pessimistic strategies prevent inconsistencies by limiting availability. Each partition makes worst-case assumptions about what other partitions are doing, and

[2] As before, by "mutual consistency" we mean that the copies contain the same value.

operates under the pessimistic assumption that if an inconsistency can occur, it will occur. These strategies differ primarily in the policy they use to restrict transaction processing. Since they ensure consistency, it is straightforward to merge the results of individual partitions; updates are merely propagated from copies in one partition to their counterparts in the other partitions at reconnection time.

At the other extreme, *optimistic strategies* do *not* limit availability. Any transaction may be executed in any partition that contains copies of the items read and written by the transaction. Hence, although transaction processing within each partition is consistent, and no user staying within a single partition would detect an inconsistency, global inconsistencies may be introduced. These strategies operate under the optimistic assumption that inconsistencies, even if possible, rarely occur. At reconnection time, the system must first *detect* inconsistencies and then *resolve* them.

Optimistic strategies differ primarily in how they detect and resolve inconsistencies. In Section 1 we discussed several ways of resolving conflicts. These include *undoing* a set of the transactions that have generated no significant external actions, running *compensating* transactions to nullify the effects of transactions generating external actions, and running *corrective* transactions that transform the database to a "correct," but not necessarily serializable, state. Obviously, the latter approach requires finding a suitable correctness criterion in lieu of one-copy serializability.

Syntactic approaches use one-copy serializability as their sole correctness criterion and check serializability by examining readsets and writesets of the executed transactions. Hence neither the semantics of the transactions (i.e., how the items read are used to generate the result) nor the semantics of the data items themselves are used in ascertaining correctness. Syntactic approaches are implemented using general-purpose concurrency control algorithms such as two-phase locking [Eswaran et al. 1976].

At the other extreme, *semantic approaches* use either the semantics of the transactions or the semantics of the database in defining correctness. Although this is somewhat of a "catchall" category, there are two discernible subcategories. The first uses serializability as the correctness criterion but also uses the semantics of the transactions to test serializability. The second abandons serializability altogether and defines correctness in terms of the contents of the database itself; the correctness criterion is intended to capture the semantics of the data stored in the database. Such semantic constraints fall outside of the traditional model of transaction processing.

3. SYNTACTIC APPROACHES

All approaches in this section use serializability as the correctness criterion and check serializability by comparing transactions' readsets and writesets. We assume that a correct concurrency-control mechanism coordinates transaction execution within a partition; hence transaction execution within a partition is serializable.

We also assume that, at the time of the partitioning, all copies are mutually consistent and there are no in-progress transactions. Note that this assumption is not realistic and is made to simplify the presentation. In general, copies of data items may not be consistent at partition time because some have processed updates of a committed transaction whereas others have not. How the system resolves these "blocked" transactions is discussed in Section 5, which deals with atomic commitment. Transactions at earlier stages of processing can be aborted and rerun in the partition containing their site of origin.

3.1 Optimistic Strategies

3.1.1 *Version Vectors* [Parker et al. 1983]

Version vectors were proposed for use in the distributed operating system LOCUS to detect write–write conflicts between copies of files [Popek et al. 1981]. Each copy of a file f has a *version vector* associated with it that counts the number of updates of f originating at each site at which f is stored. The vector consists of a sequence of n pairs, where n is the number of sites at which f is stored; the ith vector entry

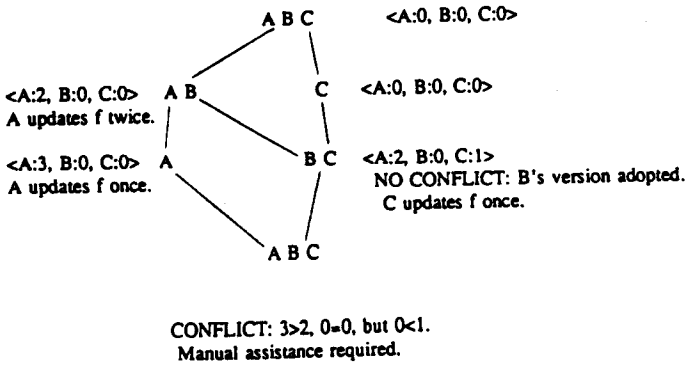

Figure 3. Conflict on file f detected by incomparable version vectors.

$(S_i : v_i)$ counts the number of updates to f, v_i, originating at site S_i. Conflicts that occur when more than one partition updates the file can be detected by comparing version vectors.

Vector $\mathbf{v}$ is said to *dominate* vector $\mathbf{v}'$ if $\mathbf{v}$ and $\mathbf{v}'$ are version vectors for the same file and $v_i \geq v_i'$ for $i = 1, \ldots, n$. Intuitively, if $\mathbf{v}$ dominates $\mathbf{v}'$, the copy with vector $\mathbf{v}$ has seen a superset of the updates seen by the copy with vector $\mathbf{v}'$. Two vectors are said to *conflict* if neither dominates. In this case, the copies have seen different updates. For example, $\langle A:3, B:4, C:2\rangle$ since $3 > 2$, $4 > 1$ and $2 = 2$, but $\langle A:3, B:1, C:2\rangle$ and $\langle A:2, B:4, C:2\rangle$ conflict since $3 > 2$ but $1 < 4$.

When two sites discover that their version vectors for f conflict, an inconsistency has been detected. How to resolve the inconsistency is left up to the database administrator (DBA).

Example. Consider the partition graph for file f shown in Figure 3. Sites A, B, and C initially have the same version of f. The system then partitions into groups AB and C, and A updates f twice. Hence both A and B have version vectors of $\langle A:2, B:0, C:0\rangle$, while C is $\langle A:0, B:0, C:0\rangle$. Site B then splits off from site A and joins site C. Since C did not update f and B has the current version, there is no conflict ($\langle A:2, B:0, C:0\rangle$ dominates $\langle A:0, B:0, C:0\rangle$), and B's version (and vector) is adopted for the new group BCE. During this new partition failure, A updates its version of f once, making group A's version vector $\langle A:3, B:0, C:0\rangle$, and C updates its version of f once, making group BC's version vectors $\langle A:2, B:0, C:1\rangle$. When groups A and BC now combine, there is a conflict and neither of $\langle A:2, B:0, C:1\rangle$ or $\langle A:3, B:0, C:0\rangle$ dominates the other.

Version vectors detect write–write conflicts only. Read–write conflicts cannot be detected because the files read by a transaction are not recorded. Hence the approach works well for transactions accessing a single file, which are typical in many file systems, but not for multifile transactions, which are common in database systems.

Example. Consider applying version vectors to the banking example of Figure 1, where communication between sites A and B fails, as shown in Figure 4. During the failure, the transaction executed at A updates the checking balance based on the value of the savings balance; the transaction executed at B updates the savings balance based on the value of the checking balance. No conflict will be detected, even though the above is clearly not serializable.

To extend the version vectors algorithm so that read–write conflicts are detectable, reads and writes of transactions must be logged. This leads to an algorithm very similar to the Optimistic Protocol presented next.[3]

3.1.2 *The Optimistic Protocol* [Davidson 1982, 1984]

The Optimistic Protocol uses a precedence graph to detect inconsistencies. A precedence graph models the necessary ordering

[3] Historical note. Such an extension was proposed by Parker and Ramos [1982]. Their conflict detection algorithm, however, is incorrect: It does not detect all inconsistencies and falsely detects inconsistencies.

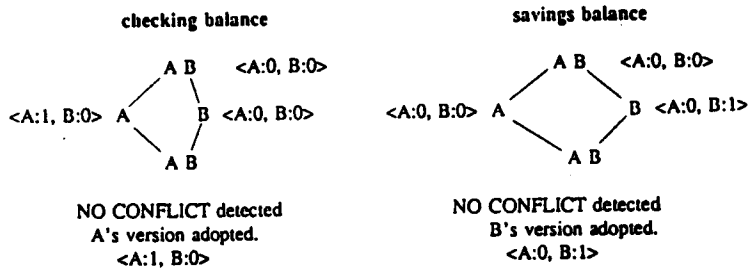

Figure 4. Incorrect conflict detection using version vectors with multifile transactions.

between transactions, and is used to check serializability across partitions. The precedence graphs are adapted from serialization graphs, which are used to check serializability within a site [Papadimitriou 1979]. In the following we assume that the readset of a transaction contains its writeset. (The reason for this assumption is to avoid certain NP-complete problems in checking serializability.)

In order to construct the precedence graph, each partition maintains a log, which records the order of reads and writes on the data items. From this log, the readsets and writesets of the transactions and a serialization order on the transactions can be deduced. (A serialization order exists since, by assumption, transaction execution within a partition is serializable.) For partition i, let $T_{i1}, T_{i2}, \ldots, T_{in}$ be the set of transactions, in serialization order, executed in i.

The nodes of the precedence graph represent transactions; the edges represent interactions between transactions. The first step in the construction of the graph is to model interactions between transactions in the same partition. Two types of edges (interactions) are identified:

(a) (*Data*) *Dependency Edges*[4] ($T_{ij} \dashrightarrow T_{ik}$). These edges represent the fact that one transaction T_{ik} read a value produced by another transaction T_{ij} in the same partition (WRITESET(T_{ij}) ∩ READSET(T_{ik}) ≠ ∅, $j < k$).

(b) *Precedence Edges* ($T_{ij} \rightarrow T_{ik}$). These edges represent the fact that one transaction T_{ij} read a value that was later changed by another transaction T_{ik} in the same partition (READSET(T_{ij}) ∩ WRITESET(T_{ik}) ≠ ∅, $j < k$).

A dependency edge from T_{ij} to T_{ik} indicates that the output of T_{ij} influenced the execution of T_{ik}; hence the "existence" of T_{ik} depends on the "existence" of T_{ij}. The meaning of a precedence edge T_{ij} from T_{ik} is more subtle: T_{ik} does not influence T_{ij} only because T_{ij} executed before it. In this case the "existence" of T_{ik} does *not* depend on the existence of T_{ij}. In both cases, an edge from T_{ij} to T_{ik} indicates that the order of execution of the two transactions is reflected in the resulting database state. Note that the graph constructed thus far must be acyclic since the orientation of an edge is always consistent with the serialization order.

To complete the precedence graph, conflicts between transactions in different partitions must be represented. A new type of edge is defined for this purpose:

(c) *Interference Edges* ($T_{ij} \rightarrow T_{lk}$, $i \neq l$). These edges indicate that T_{ij} read an item that is written by T_{lk} in another partition (READSET(T_{ij}) ∩ WRITESET(T_{lk}) ≠ ∅).

The meaning of an interference edge is the same as a precedence edge: An interference edge from T_{ij} to T_{lk} indicates that T_{ij} logically "executed before" T_{lk} since it did not read the value written by T_{lk}. An interference edge signals a read–write conflict between the two transactions. (A write–write conflict manifests as a pair of read–write conflicts since each transaction's readset contains its writeset.)

[4] Dependency edges are also called *ripple edges* [Davidson 1982, 1984].

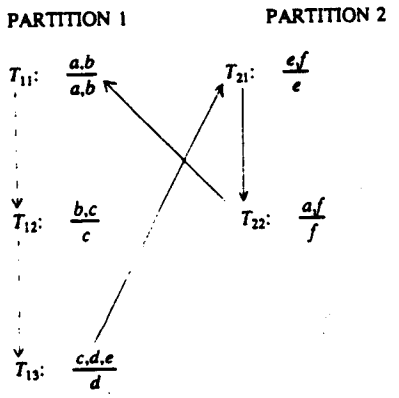

Figure 5. Conflict between transactions executed in different partitions detected by cycle in precedence graph.

Example. Suppose that the serial history of transactions executed in P_1 is $\{T_{11}, T_{12}, T_{13}\}$, and that of P_2 is $\{T_{21}, T_{22}\}$. The precedence graph for this execution is given in Figure 5, where the readset of a transaction is given above the line and the writeset below the line. (Thus, transaction T_{12} reads b, c and writes c.)

Intuitively, cycles in the precedence graph are bad: If T and T' are in a cycle, then the database reflects the results of T executing before T' and of T' executing before T—a contradiction. Conversely, the absence of cycles is good: *The precedence graph for a set of partitions is acyclic if and only if the resulting database state is consistent* [Davidson 1984]. An acyclic precedence graph indicates that the transactions from both groups can be presented by a single serial history, and the last updated copy of each data item is the correct value. A serialization order for the transactions can be obtained by topologically sorting the precedence graph.

Inconsistencies are resolved by rolling back (undoing) transactions until the resulting subgraph is acyclic. When a transaction is rolled back, transactions connected to it by dependency edges must also be rolled back, since these transactions read the values produced by the selected transaction. Hence rolling back one transaction may precipitate the rolling back of many, a problem known as *cascading rollbacks*. Transactions connected to a rolled-back transaction by precedence edges are not rolled back since they did not read the results of the rolled-back transaction. In the above example, if T_{11} is selected, then T_{12} and T_{13} must also be selected. Simply selecting T_{13}, T_{21}, or T_{22}, however, also breaks the cycle and involves only one transaction. Note that transactions must be rolled back in reverse order of execution; that is, within each partition, the value of a data item that is updated by one or more rolled-back transactions from that group will be restored to the value read by the *earliest* rolled-back transaction. To merge the partitioned databases, the final value of each updated data item in each partition group can simply be forwarded to the other group (a data item cannot be updated by both groups after transactions have been rolled back, since the resulting precedence graph is acyclic).

Note that the notion of "committing" a transaction has been somewhat violated. A transaction T is "committed" during a failure subject to confirmation at recovery. If all actions performed by T are recoverable, rolling back is not a problem; one merely replaces the values updated by T with the values read by T. However, some unrecoverable actions may also have been performed. For example, an automatic teller may have handed money to a customer, results may have been reported to a user, or a missile may have been fired. Some such actions may be compensated for; that is, there could be some T' that can be executed to nullify the effect of T. For example, the bank could charge the account of the customer who accidentally received cash from the automatic teller, or the reporting procedure could inform the user that the reported results were inaccurate due to system failure (it is hoped that the user would have been made aware of this possibility from the start). Other actions—such as the firing of a missile—may have no compensation. Such actions should not be permitted during failure since there can be no guarantee that the transaction will not be rolled back.

The algorithm used to select which transactions to roll back should strive to minimize some cost function, for example, the

number of rolled-back transactions, or the sum of the weights of the rolled-back transactions (where the assignment of weights can be application dependent). Unfortunately, minimizing either the number of transactions or the sum of their weights is an NP-complete problem [Davidson 1984]; hence heuristics must be used.

The most promising heuristics use the following observation: Breaking all two-cycles in a precedence graph tends to break almost all cycles. A *two-cycle* is a cycle consisting of two transactions connected by a pair of interference edges in opposite directions. These cycles tend to represent write–write conflicts on data items. Two-cycles can be broken optimally by using an algorithm requiring time $O(N^{2.81})$, where N is the number of transactions [Davidson 1982]. After the two-cycles have been broken, the few remaining cycles can be broken by a greedy algorithm, one that repetitively selects the lowest-weight transaction involved in a cycle. Simulation studies have shown that such heuristics work very well, outperforming all other strategies tested [Davidson 1984].

The performance of the Optimistic Protocol is studied by Davidson [1982]. A probabilistic model is developed that yields a formula for estimating rollback rate given the number of transactions, a model of the average transaction, and the size of the database. Simulation results in the same paper yield additional insight into rollback rates. These studies indicate that the Optimistic Protocol performs best when

(1) a small percentage of items are updated during the partitioning, and
(2) few transactions have large writesets.

Whenever (1) holds, the probability that a given transaction will be rolled back depends more on the size of its writeset than its readset. Concerning (2), not only is the occasional large transaction more likely to conflict with another transaction, but in addition its rollback is likely to cause other rollbacks. Consequently, the rollback rate is quite sensitive to variance in transaction size.

3.2 Pessimistic Strategies

The first group of pessimistic strategies, primary site (copy), tokens, and voting, were initially proposed as distributed concurrency-control mechanisms. However, they can also be used to prevent conflicts between transactions when the network partitions. Missing writes is an adaptive voting strategy that improves performance when there are no failures in the system. Accessible copies is an adaptation of a "read-one/write-all" protocol. The last strategy, designed specifically for partitioned networks, strives to increase availability by exploiting known characteristics of the work load.

3.2.1 *Primary Site, Copy* [Alsberg and Day 1976; Stonebraker 1979]

Originally presented as a resilient technique for sharing distributed resources, this approach suggests that one copy of an item be designated the primary copy, and as such be responsible for that item's activity. All reads for a data item must be performed at the primary site for that data item.[5] Updates are propagated to all copies. In the case of a partition failure, only the partition containing the primary copy can access the data item. Updates are simply forwarded at recovery to regain consistency.

This approach works well only if site failures are distinguishable from network failures. If this is the case and the primary site for a data item fails, a new primary can be elected (for a discussion of election protocols, see Garcia [1982]). However, if it is uncertain whether the primary failed or the network failed, the assumption must be that the network failed and no new primary can be elected.

3.2.2 *Tokens* [Minoura and Wiederhold 1982]

This approach is very similar to that above except that the primary copy of an item can change for reasons other than site failure. Each item has a token associated with

[5] Normally only the lock for a data item must be acquired at the primary site: The actual read may be performed on any copy once the lock has been granted.

it, permitting the bearer to access the item. In the event of a network partition, only the group containing the token will be able to access the item.

The major weakness with this scheme is that accessibility is lost if the token is lost as a result of site or communication medium failure.

3.2.3 *Voting* [Gifford 1979]

The first voting approach was the majority consensus algorithm [Thomas 1979]. What we now describe is the generalization of that algorithm proposed by Gifford [1979].

In this approach, every copy of a replicated item is assigned some number of votes. Every transaction must collect a read quorum of r votes to read an item, and a write quorum of w votes to write an item. Quorums must satisfy two constraints:

(1) $r + w$ exceeds the total number of votes v assigned to the item, and
(2) $w > v/2$.

The first constraint ensures that there is a nonnull intersection between every read quorum and every write quorum. Any read quorum is therefore guaranteed to have a current copy of the item. (Version numbers are used to identify the most recent copy.) In a partitioned system, this constraint guarantees that an item cannot be read in one partition and written in another. Hence read–write conflicts cannot occur between partitions.

The second constraint ensures that two writes cannot happen in parallel or, if the system is partitioned, that writes cannot occur in two different partitions on the same data item. Hence write–write conflicts cannot occur between partitions.

Example. Suppose that sites S_1, S_2, and S_3 all contain copies of items f and g, and that a partition P_1 occurs, isolating S_1 and S_2 from S_3, as shown in Figure 6a. Initially, $f = g = 0$, each site has 1 vote for each of f and g, and $r = w = 2$ for both f and g. During the partitioning, transaction T_1 wishes to update g on the basis of values read for f and g. Although it cannot be executed at S_3 since it cannot obtain a read quorum for f, or read and write quorums for g, it can be executed at S_1, and the new value $g = 1$ is propagated to S_2.

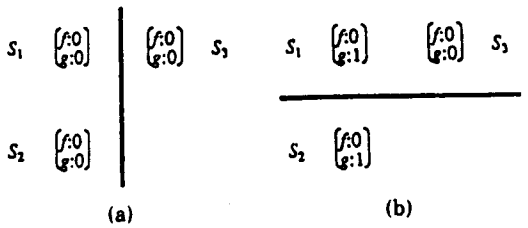

Figure 6. Correct transaction processing during partitioning using voting.

Now suppose that P_1 is repaired, and a new failure P_2 isolates S_1 and S_3 from S_2, as shown in Figure 6b. During this new failure, transaction T_2 wishes to update f on the basis of values read for f and g. It cannot be executed at S_2 since it cannot obtain a read quorum for g, or read and write quorums for f. It can be executed at S_3, however. Using the most recent copy of $g = 1$ (obtained by reading copies at both S_1 and S_3 and taking the latest version) T_2 computes the new value $f = 1$ and propagates the new value to S_1.

Notice that the above example reduces to a majority vote since each copy has exactly one vote and r and w are a simple majority [Thomas 1979].

Varying the weight of a vote can be used to reflect the needed accessibility level of an item. For example, in a banking application, a customer may use certain branches more frequently than other branches. Suppose that there are 5 branches of the bank and the customer uses branches 1, 2, and 3 with equal frequency, but never goes to branches 4 or 5. Assigning $r = w = 2$ and the customer's account at branches 1, 2, and 3 a vote of 1 but 0 elsewhere would reflect this usage pattern.

The quorum algorithm differs from those previously discussed in two important ways. First, by choosing $r < v/2$, it is possible for an item to be read accessible in more than one partition, in which case it will be write accessible in none. Read accessibility can be given a high priority by choosing r small. Second, the algorithm

does not distinguish among communication failures, site failures, or just slow response. A serious weakness of the previous schemes is that availability is severely compromised if a distinction cannot be made.

A weakness of the quorum scheme is that reading an item is fairly expensive. A read quorum of copies must be read in this scheme, whereas a single copy suffices for all other schemes.

3.2.4 *Missing Writes* [Eager and Sevcik 1983]

Eager and Sevcik's algorithm is based on the observation that while requiring a quorum for items in the readset as well as for those in the writeset is a sufficient restriction to guarantee correct or serializable execution during partition failures, it is not necessary when there are no failures [Bernstein and Goodman 1983; Eager and Sevcik 1983]. Requiring a readset quorum significantly degrades performance when there are no failures, but is necessary to guarantee correctness when there are failures. Thus transactions run in two modes, normal and failure. When in normal mode, transaction T reads one copy of each data item in its readset and updates all copies in its writeset. If some copy cannot be updated, T becomes "aware" of a missing update, and must run in failure mode. Failure mode is very similar to the majority consensus algorithm alluded to above: Quorums must now be obtained for each data item in the readset and writeset.[6] This "missing update information" is then passed along to all following transactions that need the information, that is, all transactions connected to T by a path of dependency and precedence edges originating at T. These transactions also become aware of missing updates, and must run in failure mode. Since T cannot see the future and does not know what later transactions will be affected, a level of indirection is used: Missing update information is posted at sites, along with a description of what transactions need the information. When the failure is repaired, the missing update information will eventually be posted at the sites that "caused" the missing updates, that is, those that did not receive the updates. The updates then can be applied, and postings removed from other sites throughout the system.

The algorithm hinges on the ability to recognize "missing writes" and to propagate the information to later transactions so that cycles in the precedence graph of committed transactions are avoided. Note, however, that certain transactions may be able to execute without restriction even if there are partition failures present in the system; there is no harm in allowing read-only transactions to "run in the past" during a failure, that is, to read an old value of a data item, as long as no cycles result in the precedence graph of committed transactions. This ability to run in the past allows a site that has become isolated from the rest of the network to execute read-only transactions even if updates are being performed on remote copies of the data items stored at that site.

Example. Suppose that there are four sites in the system S_1, S_2, S_3, and S_4. Sites S_1, S_2, and S_3 contain copies of data item a; site S_1, S_3, and S_4 contain copies of data item b. Now suppose that a failure occurs, isolating sites S_1 and S_2 from sites S_3 and S_4; transactions T_1, T_2, T_3 are initiated at site S_1 (in that order), while transaction T_4 is initiated at S_4. The readsets, writesets, and precedence graph are depicted in Figure 7. (The precedence graph shown is of *uncommitted* transactions since cycles in the precedence graph of *committed* transactions will obviously be avoided.)

T_1 is unaware of the failure, since it can obtain a copy of a and b at S_1; it can happily run in the past. T_2 becomes aware of the failure when it is unsuccessful at updating the copy of a at S_3; it is allowed to commit, however, since it can receive a quorum for each data item in its readset and writeset (assuming that each copy has a weight of 1). T_2 is then required to pass all of its missing update information to transactions that are incoming nodes for outgoing edges from T_2, such as T_3 in this example. If T_3

[6] A quorum can essentially be thought of as the "$w > v/2$" from Condition 2 in Section 3.2.3; it is a set of (possibly weighted) votes from sites containing copies of the data item such that any two quorums for that data item intersect.

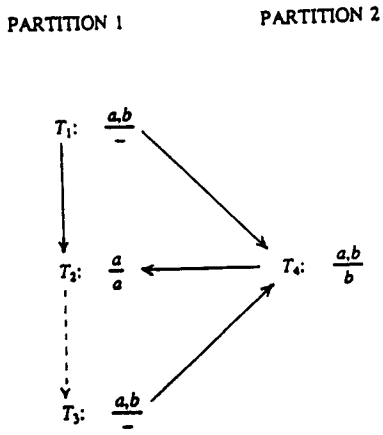

Figure 7. Potential conflict between transactions in different partitions is avoided by requiring transactions aware of missing updates to collect read and write quorums.

were to successfully commit, it would also be required to pass on the missing update information. In this example, however, T_3 is not allowed to commit; since it is aware of missing updates, it is required to obtain a quorum for data items in its readset, which it cannot for b. Transaction T_4 would also not be allowed to commit since although it can obtain a quorum for b, it finds that it cannot update the copy of b at S_2, and must then run in failure mode. Since it cannot obtain a quorum for a, it cannot complete successfully. Thus in this example (as well in all others), there are no cycles in the precedence graph of *committed* transactions. Note that the restriction that T_2 and T_4 be rerun in failure mode is necessary. Suppose that T_2 and T_4 both read a and b, but T_2 updated a while T_4 updated b. If they both executed in normal mode and did not switch to failure mode when they become aware of missing updates, a cycle would result in the precedence graph of committed transactions.

In order to implement this method, regardless of the concurrency-control mechanism being used, several files must be kept at each site. They include

(a) a file for posted missing updates, with indications of which transactions need to be informed about the missing updates;
(b) a file containing the values of missing updates, to be applied to the appropriate copies when recovery occurs;
(c) a file indicating the transaction restarts, aborts, or commits of which the site is aware, used to resolve the "blocked" transactions alluded to in the introduction to Section 3;
(d) a record of the missing updates that have been applied at the site.

Although these files can grow very rapidly if the system is active during failures, they must only be maintained when failures are present in the system, and thus do not impact performance in the absence of failures. Furthermore, since quorums are only required when a transaction is aware of a missing update, when there are no failures or the transaction is not required to know about the failure, reading an item incurs no additional overhead. The method is also very flexible: It requires no "detection" of failure other than the inability to perform updates, and no special "global" action or temporary cessation of activity to propagate updates when the failure is repaired.

3.2.5 *Accessible Copies Algorithm* [El Abbadi et al. 1985]

The Accessible Copies algorithm is based on the following intuitive, "read-one/write-all" protocol:

(1) A data item can be read and written within a partition only if a majority of its copies reside on member sites of the partition. In this case, the item is said to be *accessible*.
(2) A read operation on an accessible data item is implemented by reading the nearest copy of the item residing on a member of the partition.
(3) A write operation on an accessible data item is implemented by writing all copies residing on members of the partition.

The first rule of the protocol ensures that only one partition may access a given data item. The second and third rules ensure that the copies of a data item remain consistent within a partition.

The above protocol is appealing because it is simple and because it implements the read operation inexpensively. The protocol ensures one-copy serializability in an "ideal" network, where partition failures are "clean" and sites detect partition failures almost instantaneously. Unfortunately, if either property of the ideal network is violated, which sometimes happens in any real system, incorrect executions can occur.

The principal idea in the Accessible Copies algorithm is the implementation of an abstract communication layer on top of the real communication network, where the behavior of the new layer approximates that of the "ideal" network. A variant of the above read-one/write-all protocol can then be implemented on top of the abstract communication layer.

The abstract communication layer creates and manipulates *virtual partitions*, which are rough analogs of the actual partitions that occur in the real network. A virtual partition has three important attributes. The first is its *creation time*, which is the logical clock time of its creation [Lamport 1978]. The second is its *set of potential members*, which is the set of sites that are allowed to join the partition. The third is its *set of actual members*. The first two attributes are static, and are known to each member of a virtual partition. The third attribute is dynamic, and generally will not be known with certainty by any site in the virtual partition.

One important difference between real and virtual partitions is that virtual partitions are created explicitly according to a well-defined protocol. Loosely speaking, the steps of the creation protocol are as follows. First, a group of sites depart from their current virtual partitions. (A site can depart from its current virtual partition unilaterally by setting a local variable.) Second, the group of sites collectively determine the creation time and the potential members of the new virtual partition. The creation time must be larger than any previous creation time, and the set of potential members can include only those sites participating in the creation protocol. Last, the sites in the group asynchronously become actual members of the new virtual partition (be setting an appropriate variable). It should be remarked that the creation protocol given by El Abbadi et al. [1985] tolerates additional partition failures occurring during its execution.

Given a correct implementation of the abstract communication layer, a variant of the simple read-one/write-all protocol can be used to control access to data items. The variant protocol is obtained by substituting the phrase "potential member(s)" of the virtual partition for all occurrences of the phrase "member(s) of the partition" in the original protocol. The resulting protocol provides one-copy serializability when used in conjunction with an appropriate failure recovery protocol.

3.2.6 *Class Conflict Analysis* [Skeen and Wright 1984; Wright 1983]

The pessimistic strategies discussed thus far strive to make each data record available for reading and writing in some partition by arbitrary transactions. These strategies, then, emphasize the general availability of individual records. An alternate strategy, class conflict analysis, strives to ensure the capability of performing important *high-level operations* on the data. Hence this strategy emphasizes the availability of high-level data operations, possibly at the expense of the general availability of records.

To illustrate the difference between the two approaches, consider again the banking example shown in Figure 2, where a customer can overdraw his or her checking account as long as the overdraft is covered by funds in his or her savings account. If the system partitions, none of the discussed pessimistic strategies would allow a checking withdrawal (which requires reading the balance of both accounts) to occur in one partition and allow a savings deposit to occur in another partition. However, executing these transactions in parallel in different partitions violates neither the bank's policy nor the one-copy serializability. Hence these transactions should be allowed.

The class conflict analysis approach assumes that transactions are divided into classes as proposed in SDD-1 [Bernstein et

al. 1980]. A class may be a well-defined transaction type, such as the "savings withdrawal," or it may be syntactically defined, for example, the class containing all transactions reading and writing a subset of items *a*, *b*, and *c*.

Like transactions, classes are characterized by their readsets and writesets. The readset of a class is the union of the readsets of all of its member transactions; similarly, the writeset of a class is the union of the writesets of all its member transactions. As before, it is assumed that a class's readset contains its writeset, so that NP-complete problems are avoided. Two classes *conflict* if one's readset intersects the other's writeset. A class conflict indicates a *potential* read–write conflict between member transactions of the classes. (A conflict may not actually occur because the transactions' readsets and writesets may be proper subsets of the classes' readsets and writesets.)

When a failure occurs, each partition group must decide what classes of transactions it will execute so as to avoid potential conflicts with transactions executed in other partitions. As a first step, it must decide what classes are "assigned" to its partition as well as those that are assigned to the other partitions. For example, if classes are executable at specific sites, the classes assigned to a partition would be those executable at sites within the partition. Note that classes may be assigned to more than one partition, and there may be conflicts between classes in different partitions.

The second step for each partition is to analyze the assignment and discover the class conflicts that can lead to nonserializable executions. The analysis uses a graph model similar to the precedence graph used in the Optimistic Protocol, except that where precedence graphs give the *actual* orderings between conflicting transactions, class conflict graphs give all *potential* orderings between conflicting classes. A simplified version of the model is defined below.

A node of the *class conflict graph* represents the occurrence of a given class in a given partition. Edges are drawn between occurrences of conflicting classes according to the rules given below. Let C_i and C_j be classes such that READSET(C_i) ∩ WRITESET(C_j) is not empty.

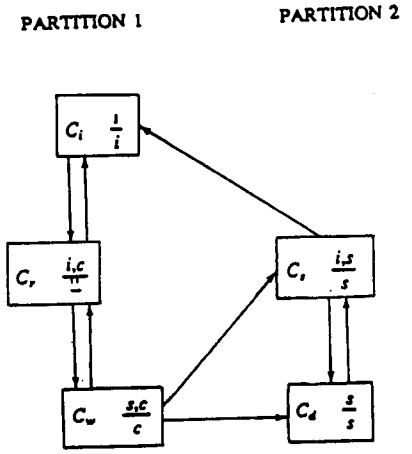

Figure 8. Potential conflict indicated by multipartition cycles in class conflict graph.

(1) If C_i and C_j are in the same partition, then a pair of edges pointing in opposite directions connects them.
(2) If C_i and C_j are in different partitions, then a directed edge extends from C_i to C_j.

The direction of the edges indicates the possible logical orderings of transactions from conflicting classes. In particular, in the case of classes C_i and C_j in Rule (2), the transactions of C_i cannot logically succeed those of C_j because C_i's transactions cannot read the updates of C_j's transactions. Therefore the only order possible is that all transactions of C_i precede all transactions of C_j, as indicated by the single directed edge.

Example. Figure 8 is a class conflict graph for the banking example for two partitions. Boxes denote classes. Readsets are shown above the line, and writesets, below. Data items *s*, *c*, and *i* are the savings account, the checking account, and the interest rate, respectively. Classes C_d and C_w include the savings deposit transactions and checking withdrawal transactions discussed in Section 1. Class C_i transactions change the interest rate, class C_s transactions add an interest payment to the

savings account, and class C_r transactions are read only.

The third step in the analysis is to identify those assignments that could lead to nonserializable executions. Cycles play a key role here, but not all cycles are bad. Among class occurrences in the same partition, cycles are both common and harmless, since the concurrency control algorithm operating in the partition will prevent nonserializable executions. On the other hand, cycles spanning multiple (>1) partitions are not harmless, since there is no mechanism preventing them in an execution. Hence *multipartition cycles indicate the potential for nonserializable executions.* In the example, if transactions from classes C_i, C_r, and C_w execute in that order in partition 1 and a transaction from C_s executes in partition 2, the result is serializable. (This can be checked by constructing the precedence graph for the execution.)

Whenever the preliminary class assignment yields a (multipartition) cyclic graph, further constraints on transaction processing must be imposed. The most straightforward approach is to delete classes from partitions until the class conflict graph is rendered multipartition acyclic. In the above example, one of C_i, C_r, C_w, or C_s must be deleted. For availability reasons, it is desirable to delete a minimum set of classes. Not surprisingly, this is an NP-complete problem.

Although this discussion has assumed that the complete state of the network is known to all partitions, this assumption is not required in applying class conflict analysis. Wright discusses some modifications to the basic algorithm that work with incomplete knowledge of the network status and some refinements that afford more availability than the analysis presented here [Wright 1983].

3.3 Discussion

3.3.1 Optimistic versus Pessimistic

An appropriate cost model is one basis for comparing the two approaches. The model should include *overhead*, the cost of *repairing inconsistencies*, and the cost of *lost opportunities*. In the following, costs common to all approaches, such as the propagation of updated values, are omitted.

Optimistic policies have two sources of overhead. The first is the log, which must be maintained while the system is partitioned, recording the readset and writeset of each transaction in order to construct the precedence graph, and recording sufficient information to roll back transactions. Many database systems already maintain a log, called an *undo log*, for rolling back transactions in case of site failures or transaction failures (e.g., deadlocks) [Gray et al. 1981]. This same log can be used to roll back conflicting transactions in a partitioned system. In order to construct the graph, however, undo logs must be augmented with records of transactions' readsets (which are normally not recorded since they are not needed to roll back a single transaction). This increases the complexity of the logging algorithms, but it does not significantly increase the cost of logging in most systems.

The second and most significant source of overhead in optimistic strategies is the conflict detection algorithm, which constructs the graph, checks the graph for cycles, and then selects transactions to roll back. Graph construction requires a single pass over the entire log, which can be quite expensive for a partition of long duration. The selection algorithm can be made arbitrarily expensive, depending on the quality of heuristics used. As mentioned in the description of the Optimistic Protocol, the best heuristics require time $O(N^{2.81})$, where N is the number of transactions. However, linear time heuristics often yield acceptable solutions.

The cost of repair in an optimistic approach is simply the rollback rate times the cost of rolling back a transaction. We have already discussed rollback rate. The rollback cost is often a significant fraction of the transaction's execution cost, and may, in fact, exceed the execution cost if the transaction has external side effects (e.g., a customer may be entitled to compensation if his or her reservation is canceled,

or a series of transactions may need to be executed to compensate for a single rolled-back transaction). Consequently, the rollback rate must be kept reasonably small (certainly less than 20 percent) if optimistic approaches are to be cost effective.

The goal of optimistic approaches is to minimize lost opportunity, the cost associated with needlessly delaying a transaction. These costs can be substantial when user satisfaction is important as, for example, in a banking application. Lost opportunities still occur in these approaches because of the allocation of resources to transactions that are destined to be rolled back. Such transactions may displace valid transactions during the partitioning, and rolling them back may cause further delays after the partitions are reconnected. Still, for most applications, we speculate that other costs dominate.

Pessimistic approaches have no repair costs and, except for class conflict analysis, almost no overhead. Even in class conflict analysis, the overhead is likely to be substantially less than in an optimistic strategy, because although conflict analysis and conflict detection are procedurally similar, the number of predeclared classes in conflict analysis is likely to be substantially less than the number of transactions in conflict detection.

The major cost of a pessimistic approach is, of course, the cost of lost opportunities. Included in this cost are not only opportunities lost to real partitioning but also opportunities lost to "apparent" partitionings, for example, site failures that are indistinguishable from real partitionings. In many systems, apparent partitionings occur more frequently than real partitionings; therefore they must be included in any cost analysis.

In summary, the cost of an optimistic strategy is the overhead of conflict detection plus the repair cost, whereas the cost of a pessimistic strategy is the cost of opportunities lost to real and apparent partitionings. Unfortunately, except for repair costs, informed estimates for these costs are not easily obtained. No one has measured the overhead associated with any of the strategies, and the cost of lost opportunities is hard to quantify (although one component in a pessimistic strategy is the cost of underutilization of processing resources).

3.3.2 Combining Strategies

Instead of using one strategy during a partitioning, strategies can be combined vertically over time; the system could start out using one strategy and switch to another as circumstances dictate. For example, the number of transactions rolled back in the Optimistic Protocol has been observed to increase roughly quadratically with time. In fact, the expected number of transactions rolled back can be estimated with a formula involving the number of transactions processed within the partition, the number of data items in the database, and certain other parameters modeling the type of transactions being executed [Davidson 1982]. Since it is usually impossible to predict how long a partitioning will last, the database administrator could then set a ceiling on the rollback rate (say 10 percent) and request that the Optimistic Protocol be used only until this ceiling were reached. If this ceiling was reached, the system could switch to a more pessimistic approach, such as primary site, for the remainder of the failure. Of course, there is no guarantee that the subsequent transactions would not also be rolled back since they could be connected by dependency edges to transactions that had already executed. These transactions would still have to be included in the construction of the precedence graph, and considered for possible rollback, to guarantee serializability. The rollback rate, however, would be held at a more acceptable level.

Strategies can also be combined horizontally over time [Skeen 1982c]. One approach is to assign items different levels of consistency. Items in level 0 (the highest level) are immutable during a partitioning, items in level 1 are updated according to a pessimistic strategy, and items in level 2 are updated according to an optimistic strategy. Updates to level 1 items are globally consistent and guaranteed to persist,

whereas updates to level 2 items are consistent within the partition but may not be globally consistent and, hence, are subject to rollback. Although a transaction may update items in only one level, it may read items of the same level and higher.

Another way to combine approaches horizontally is to divide transactions, instead of items, into groups. For each partition, transactions are divided into two groups: high-priority transactions that cannot be rolled back and low-priority transactions that can. Class conflict analysis is used to determine a group of high-priority transactions for each partition. The low-priority group for a partition consists of all transactions not writing an item read by a high-priority transaction in the same partition. (A low-priority transaction, however, can write an item read by a high-priority transaction in a *different* partition.) When partitions are reconnected, the Optimistic Protocol is used to construct a precedence graph containing all transactions; however, only low-priority transactions are liable to rollback. (An approach similar to this is used by Apers and Weiderhold [1984].)

4. SEMANTIC APPROACHES

The first three approaches presented in this section are optimistic, and illustrate different ways of using semantics to decrease conflict. The first approach, log transformations, uses the standard notion of correctness (serializability) but uses the semantics of transactions to check serializability. The second approach, weak consistency, slightly relaxes the standard notion of serializability in order to enrich the set of transactions allowed in a partitioned system, and uses the semantics of the application to determine when serializability can be relaxed. The third approach, Data-Patch, abandons serializability altogether, and uses an application-specific definition of correctness instead. The last approach, general quorum consensus, is pessimistic. A new correctness criterion is defined on the basis of an abstract data type definition of data items, and type-specific information is used to increase the availability of data.

This section ends with a brief discussion of some other proposed ideas for increasing availability.

4.1 Optimistic Strategies

4.1.1 *Log Transformations* [Blaustein et al. 1983]

This approach is similar to the Optimistic Protocol. During the partitioning, logs are kept of which transactions were executed and in what order. After reconnection, a rerun log is constructed, which indicates what should be reflected as having happened during the failure. To achieve this, transactions in each group may have to be rolled back and rerun. It differs in that transactions are predefined, and semantic properties of pairs of transactions are declared to avoid needlessly rolling back and rerunning transactions. These properties can include commutativity ($T_iT_j = T_jT_i$) and overwriting ($T_iT_i = T_j$). There is also a notion of "absolute time" in each group during the failure so that transactions can be merged based on the time at which they were executed.

Example. Suppose that during a partition, P_1 has executed T_2, T_4, T_6 and that P_2 has executed T_1, T_3, T_5, where the subscripts indicate the absolute timing of the transactions. The rerun log would be $T_1, T_2, T_3, T_4, T_5, T_6$. If we ignored any semantic properties of transactions, merging the database at P_1 would involve rolling back transactions T_2, T_4, T_6 and reexecuting the rerun log. If we assume that rolling back transaction T can be achieved by running an inverse transaction T^{-1}, then the entire merging operation at P_1 can be represented by the rollback log T_6^{-1}, T_4^{-1}, T_2^{-1}, followed by the redo log. Similarly, the merge operation at P_2 involves executing the rollback log T_5^{-1}, T_3^{-1}, T_1^{-1}, followed by the redo log. Let us call the combined rollback, redo log the *merge log.*

If we know that T_1 commutes with T_2, then the merge log at P_1 can be reduced to

$$T_6^{-1}, T_4^{-1}, T_1, T_3, T_4, T_5, T_6.$$

To see that the result of executing P_1's merge log is equivalent to the result of

executing $T_1, T_2, T_3, T_4, T_5, T_6$ in order, consider the entire sequence of transactions executed by P_1 (i.e., the original execution followed by the merge log):

$$T_2, T_4, T_6, T_6^{-1}, T_4^{-1}, T_1, T_3, T_4, T_5, T_6.$$

Since T_6, T_6^{-1} and T_4, T_4^{-1} are equivalent to the null transaction, the above is equivalent to

$$T_2, T_1, T_3, T_4, T_5, T_6.$$

By the commutativity of T_1 and T_2 this is equivalent to the desired sequence.

If in addition we know that T_1 and T_3 commute with T_4 and T_6, and that T_6 overwrites T_5, then the P_1 merge log can be further reduced to

$$T_1, T_3$$

(i.e., after the partition we only have to run T_1, T_3 without rolling back any transactions). At P_2, this same semantic information only reduces the merge log to

$$T_5^{-1}, T_3^{-1}, T_2, T_3, T_4, T_6.$$

The process of reducing the size of the merge log is called *log transformation*. The process can be automated with the aid of a graph formalism, which represents merge logs as graphs, and performs each log transformation as a graph transformation [Blaustein et al. 1983].

One advantage of log transformations is that merge processes at the different sites are independent of each other. That is, as each site finds out about transactions that were executed elsewhere, it can proceed to integrate them locally, regardless of what the other sites are doing. This idea has been used by Sarin et al. [1985] to extend log transformations as a general mechanism to achieve mutual consistency without guaranteeing serializability; network partitions and site failures do not have to be detected [Sarin et al. 1985]. A total ordering is imposed on updates using timestamps. Each site's data copy is only required to reflect the updates seen by the site, executed in timestamp order. If an out-of-sequence update is received (i.e., one whose timestamp is less than the timestamp of the most recent update seen by the site), log transformations are used to achieve the correct value for the copy. Serializability is not guaranteed since the copies read by a transaction may only reflect some incomplete subset of updates to the data item.

This approach may be useful in an environment where failures are common and communications unreliable.

4.1.2 *Weak Consistency* [Garcia and Weiderhold 1982]

Garcia and Weiderhold [1982] argue that conventional correctness criteria—in particular, serializability—may be stronger than needed for many read-only transactions. Since such transactions do not change the database state, their execution cannot generate inconsistencies. Relaxing the serializability constraint is especially attractive for partitioned systems, since it would allow a richer mix of read-only transactions. (The original motivation for a weaker correctness criterion was to speed up the processing of read-only transactions in a distributed system.) Since read-only transactions occur frequently in most systems, by allowing a richer mix of them one can substantially increase the number of transactions executed while partitioned.

Read-only transactions are divided into two classes: those requiring strong consistency and those requiring weak consistency. A strongly consistent transaction is processed in the normal fashion: Its execution must be serializable with respect to update transactions and other strongly consistent transactions. A weakly consistent transaction must see a consistent database state (the result of a serializable execution of update transactions), but its execution need not be serializable with respect to other read-only transactions. (Weak serializability is stronger than degree 2 or 1 consistency as defined by Gray et al. [1976]. Specifically, with degree 2 or 1 consistency, a read-only transaction can obtain an inconsistent view of the database.) The following example illustrates this.

Example. Consider again the banking database of the first section with sites A and B partitioned. The sequence of transactions given in Figure 9 occurs. Notice that the two update transactions, considered

	SITE A		SITE B
C:	*checking deposit of* $50	D:	*savings deposit of* $100
A_A:	*read checking and savings accounts*	A_B:	*read checking and savings accounts*

Figure 9. Nonserializable transaction execution allowed with weakly consistent transactions.

alone, are serializable. In fact, since they access different items, both C ; D and D ; C are valid serialization orders. However, when the accounting transactions A_A and A_B are included, the execution is not serializable. The database state read by A_A is possible only if C executes before D, whereas the state read by A_B is possible only if D executes before C. (Both A_A and A_B see a valid serialization order of the updates; the problem is that they see different orders.)

If A_A and A_B required only weak consistency, the above execution would be "correct": The update transactions alone are serializable, and each weakly consistent transaction sees the result of a serializable execution of update transactions.

The use of different consistency levels can be integrated with any of the syntactic approaches discussed in the previous section. In a pessimistic strategy, a transaction requiring only weak consistency can be executed at any time in any partition, as long as the partition contains copies of items read by the transaction. The transaction will always see a consistent database state since all update transactions are guaranteed to be (globally) consistent. In an optimistic strategy, such a transaction sees a consistent state only if it does not read the result of an update transaction that is eventually rolled back.

The choice of a consistency level for a read-only transaction depends on how the information returned by the transaction is used. An accounting transaction reporting cash flow within a bank probably requires strong consistency. Inventory reporting and transactions computing summary statistics often need only weak consistency.

Fischer and Michael [1982] give an important application of weak serializability in their algorithms for directory systems. A directory supports only three types of transactions: insert a unique item, list all items, and delete an item. Mail systems, calendar systems, and other familiar applications can be cast as directories. Exploiting the property that the list operation requires only weak consistency; they give an algorithm allowing unrestricted transaction processing in the presence of communication failures, including but not limited to failures partitioning the system.

4.1.3 *Data-Patch* [Garcia et al. 1983]

Data-Patch is a tool that aids the DBA in developing a program to automatically integrate divergent databases. As in the previous optimistic strategies, transactions are executed "normally" during the failure. At reconnection, the final database state is constructed according to an integration program. Serializability is no longer the correctness criterion; rather, the integration program defines the "correct" final database. This is based on the premise that users may already have observed the effects of a nonserializable execution; thus restoring the database to a serializable state may not be the most sensible thing to do. For example, in an airline reservation system, if a flight becomes overbooked, it may not be desirable to cancel reservations since a promise has been made to customers and normal passenger cancellations could take care of the problem.

The major design principle involved is identifying *image* and *plan* relations. Image relations are observable entities or relationships, and must reflect that in the final database. For example, in a database for Girard bank, the relation GIRARD(BRANCH, CASH, . . .) might be used to record the amount of cash at each branch. The value of CASH in each tuple at recovery should reflect the actual amount of cash at that branch. This might be obtained as the latest value for CASH in each partition group. Plan relations do not represent observable entities, and the DBA can therefore have more freedom in

selecting the final values. In the next example, ACCOUNT is a plan relation.

Example

ACCOUNT (CUSTOMER, BALANCE, . . .)
DEPOSIT (CUSTOMER, AMOUNT, DATE, . . .)
WITHDRAWAL (CUSTOMER, AMOUNT, DATE, . . .)

DEPOSIT and WITHDRAWAL are records of account activity. If during a partition a customer overdraws his or her account according to the records from each group, he or she may be assessed a penalty charge. Thus BALANCE would reflect the sum of withdrawals and deposits to the account, plus the penalty charge. If, on the other hand, a customer is mistakenly assessed a penalty charge because a DEPOSIT did not appear during a failure, the penalty charge may be dropped.

The above example shows that not only must a final database state be chosen, but corrective actions must also be specified. That is, if integrity constraints are violated after the image and plan relations have been constructed, some sort of compensating or corrective action must be issued (e.g., a penalty for overdraft, as above).

The Data-Patch integration program is defined through a set of rules that specify how the integrated database can be obtained from two databases that exist after a partition. Some rules specify how differing facts are to be combined. For example, consider a field that represents the location of a ship. In this case, the DBA can select a "latest value" rule: If the field has a different value in each partition, use the value with the latest timestamp in the integrated database. If the field represents the number of reservations for a flight, the "arithmetic rule" can be used: The integrated value is the sum of the two partitioned values minus the value that existed when the partition started. Other rules specify the corrective actions to be taken. For instance, a rule might specify that if the withdrawals exceed the deposits to an account (after the integrated database has been obtained), then a dunning letter should be sent to the customer.

4.2 Pessimistic Strategies

4.2.1 General Quorum Consensus [Herlihy 1984]

As the name suggests, general quorum consensus extends the quorum voting algorithm proposed by Gifford [1979] (see Section 3.2). This approach to replicated data management uses type information to increase data availability. In addition, it uses a novel correctness criterion and representation of data items.

Each data item is viewed as an instance of an abstract data type. An abstract data type defines the set of operations supported by items of that type. For example, a FIFO (first-in, first-out) queue defines the operations ENQUEUE (append an element to the end of the queue) and DEQUEUE (remove an element from the head of the queue and return its value). Items of type queue can only be accessed (read or written) through the ENQUEUE and DEQUEUE operations.

In addition to defining a set of operations, an abstract data type defines a type-specific correctness criteria. This correctness criteria consists of two parts: a serial specification and a behaviorial specification. The *serial specification* describes the sequences of operations that are allowed on data items of the given type. In the FIFO queue example, a sequence of ENQUEUE and DEQUEUE operations is allowed only if the number of ENQUEUE operations is greater than or equal to the number of DEQUEUE operations in any prefix of the sequence. This ensures that DEQUEUE is never applied to an empty queue. The *behavioral specification* describes the conflicts between operations that limit concurrency. For example, although ENQUEUE operations may always be performed, DEQUEUE operations conflict with both ENQUEUE operations and other DEQUEUE operations since the value returned by a DEQUEUE operation depends on the contents of the queue.

An interesting aspect of Herlihy's approach is that each data item is represented by its *history*, that is, the sequence of operations that have been applied to the data item since its creation. Each operation in the history has an associated *timestamp* that serves to uniquely identify the operation and determine its position in the history. For example, the history for some queue Q might contain the following operations (with timestamps):

Q: ⟨(ENQUEUE(3), 10036), (ENQUEUE(7), 10072), (DEQUEUE( ), 10137), (ENQUEUE(5), 10201), (DEQUEUE( ), 21007), (ENQUEUE(5), 22137)⟩

This sequence of operations results in a queue containing two elements, both with value 5.

Each copy of an item stores a subsequence, or *subhistory*, of the item's entire history. A copy's subhistory is often incomplete owing to failures preventing the receipt of certain operations. It is straightforward to merge subhistories of two or more copies into a single, more complete subhistory, however. In fact, the advantage of representing items by their histories over representing items by their values is that incomplete subhistories can be merged, whereas incomplete values typically cannot.

The basic idea behind the general quorum consensus algorithm is to associate a read quorum and write quorum with each operation defined by an abstract type. For operation OP, let R_{OP} denote its read quorum, and W_{OP} denote its write quorum. The execution of an operation OP on data item D consists of three steps:

(1) The site executing OP requests subhistories from at least R_{OP} copies of D. If less than R_{OP} copies respond, the operation cannot be executed.
(2) The executing site merges the subhistories received into a more complete subhistory. Using this merged subhistory and the serial specification for the type of D, it checks whether OP is allowed and computes the value to be returned to the user (if any).
(3) The executing site sends OP to at least W_{OP} copies. The copies append OP and its timestamp to their subhistories.

The subhistory constructed in Step (2) may be the complete history of the data item, but this is not always required. What is required is that the constructed subhistory contain enough information to determine whether OP is allowed and to determine the value returned to the user.

The choice of read and write quorum sizes is determined by the behavioral specification of the data type. If the behavioral specification indicates that operation $O1$ can influence either the acceptability or the return value of operation $O2$, then the quorum sizes must be chosen so that $W_{O1} + R_{O2}$ exceeds number of copies of the data item. In this case the quorums are said to intersect. This constraint ensures that the subhistory constructed in Step (2) during an execution of operation $O2$ will contain all prior executions of operation $O1$.

Example. In the queue example, each DEQUEUE and ENQUEUE operation influences the value returned by a later DEQUEUE operation. Hence the write quorums of both ENQUEUE and DEQUEUE must intersect the read quorum of DEQUEUE. On the other hand, neither ENQUEUE nor DEQUEUE operations influence later ENQUEUE operations. Consequently, the read quorum of ENQUEUE need not intersect with any other operation's write quorum. Given this information, one possible assignment of quorums for a queue with three copies is

$$R_{\text{ENQUEUE}} = 0,\ W_{\text{ENQUEUE}} = 1;$$

$$R_{\text{DEQUEUE}} = 3,\ W_{\text{DEQUEUE}} = 3.$$

Note that this assignment allows ENQUEUE operations to occur concurrently in different partitions. Other quorum voting schemes (namely, Gifford's) would not allow this.

General quorum consensus is an elaborate protocol. Many extensions of the above basic (and somewhat oversimplified) scheme are discussed by Herlihy [1984, 1985]. One particularly interesting exten-

sion allows quorums to be reassigned dynamically, according to detected failures and recoveries. Recall that a similar technique was used in the Missing Writes Protocol [Eager and Sevcik 1983].

4.3 Other Ideas

Numerous ad hoc techniques for exploiting the semantics of an application to increase availability have been proposed. Many of these can best be illustrated by examples.

The first example illustrates the idea of *splitting* a data item [Hammer and Shipman 1980]. In an airline reservation system, let *SEATS* represent the number of seats available on a particular flight. When a partition occurs, P_1 creates $SEATS_1$ containing 40 percent of the value of *SEATS*, and P_2 creates $SEATS_2$ containing 60 percent of the value of *SEATS* (or other percentages reflecting the relative booking rates for that flight). At recovery,

$$SEATS = SEATS_1 + SEATS_2$$

would restore *SEATS* to its correct value.

The second example comes from Incomplete Information Systems [Davidson 1982; Lipsky 1979]. Suppose that we have a tuple representing John Doe's age as less than 30. During a partition, P_1 gathers more information and concludes that his age is between 20 and 30, while P_2 concludes it to be between 15 and 25. At recovery, the intersection of these ranges, 20 to 25, may be taken as John Doe's age.

The last example illustrates the use of *failure-mode integrity constraints*. Failure-mode integrity constraints are constraints that are only checked when the system is partitioned. Recall the banking example of Figure 2, where overdrafts on the checking account were allowed as long as *checking balance* + *saving balance* ≥ 0. The example described a scenario where this constraint was violated during a partitioning. This anomaly could have been avoided by enforcing a failure-mode integrity constraint disallowing checking account overdrafts when the system is partitioned.

These ideas can be used with a pessimistic approach such as primary copy to allow more transactions to be executed: A portion of *SEATS* would be available in each group, although the actual or current value for *SEATS* could not be obtained because of possible bookings in the other group. The flight would never be overbooked, however, if neither group sold more than their allotment of seats. It can also be used with optimistic approaches such as the Optimistic Protocol and Data-Patch to avoid conflict and possible transaction rollbacks. In the Optimistic Protocol, conflicts are mainly caused by updates to the same data item. By splitting data items and recombining at recovery, this can be avoided. In Data-Patch, integration becomes easier since the value for *SEATS* can simply be computed without canceling reservations.

5. ATOMIC COMMITMENT

A transaction on a distributed database typically executes at several sites. In order to ensure the "all or nothing" property of the transaction, the executing sites must unanimously agree to commit or to abort the transaction. Until now we have assumed that this agreement, known as atomic commitment, can be achieved in a partitioned system. Let us now examine how reasonable this assumption is.

Viewed abstractly, in a commitment protocol each participant first votes to "accept" or "reject" the transaction according to its ability to process the transaction and then decides whether to commit or abort based on the voting. Commitment normally requires unanimous acceptance.[7] Of course, all decisions must agree.

The *two-phase commit protocol* is a straightforward implementation of the above [Gray 1978]. In the first phase, a designated participant, the coordinator, solicits the votes from its cohorts. In the second phase, it decides on the basis of the votes and then sends the decision to all participants. In the course of the protocol, each participant voting "accept" goes through three distinct states: an *uncommitted* state in which it has not voted, an *in*

[7] Some protocols for fully replicated databases require only acceptance by a majority.

doubt state in which it has voted but does not know the result of the voting, and a *decision* state in which it knows the commit/abort decision. (A participant voting "reject" does not occupy the in doubt state since it knows the eventual outcome.)

Consider the consequences of a partitioning occurring during the execution of the two-phase commit protocol. In each partition the participants, acting together, will attempt to decide the outcome on the basis of their states. If the partition contains the coordinator, a decided participant, or an uncommitted participant, a consistent decision can be reached (in the case of an uncommitted participant, abort will be chosen). However, a partition containing only in-doubt participants and lacking the coordinator cannot safely decide: The participants cannot commit since they do not know the outcome of the voting, and they cannot abort since they may contradict the decision of the coordinator. Hence these sites must wait until reconnection before deciding, and the protocol (and associated transaction) is said to be *blocked* at those sites.

Given that the two-phase commit protocol occasionally blocks, the interesting question then is: Are there any nonblocking protocols for partitionings? The answer is no: Even under the most favorable, realistic partitioning assumptions, there are no nonblocking protocols [Skeen 1982b]. The situation is even worse if sites can fail during a partitioning; in this case there is no protocol that guarantees that even a single site will be able to decide.

Since it is impossible to eliminate blocking, it is desirable to minimize it. Several protocols have been proposed that, under appropriate partitioning assumptions, block less than the two-phase commit protocol. One protocol, the *decentralized two-phase commit protocol*, reduces the likelihood of blocking by decreasing the time a site spends in the in doubt state [Skeen 1982c]. This is accomplished by having the participants send their votes directly to each other, bypassing the coordinator. Another protocol, the *quorum commit protocol*, reduces the probability that a large partition (one consisting of many participants) will be blocked in the event of a partitioning by introducing extra phases [Skeen 1982a, 1982b, 1983]. Its principal advantage is that it is also resilient to site failures and (nonpartitioning) communication failures. Both protocols have drawbacks, however. Although the decentralized protocol decreases the probability that a partitioning will occur while sites are in the in doubt state, it increases the expected number of blocked sites if a partitioning should occur. The quorum protocol actually increases the chance that some site will be blocked in the event of a partitioning (although the expected number of blocked sites decreases).

How the partition strategies discussed in Sections 3 and 4 treat blocked transactions depends on whether the strategy is pessimistic or optimistic. In a pessimistic strategy, the data items at undecided sites must be rendered inaccessible until reconnection. In an optimistic strategy more flexibility is possible. A partition can tentatively commit or abort a blocked transaction. If its decision is inconsistent with other decisions, it can resolve this in the same way that it resolves other inconsistencies, by rolling back the offending transaction and all dependent transactions. Since rolling back is fairly expensive, a tentative decision should be made only if it has a high probability of being correct.

6. CONCLUSION

6.1 Guidelines for Selecting a Partition Strategy

Research in distributed databases has been criticized for devising strategies for isolated problems [Mohan 1980]. In particular, concurrency control mechanisms and partition failure protocols are highly interdependent and should not be considered in isolation from each other. For example, a voting partition failure protocol should not be used with a primary site concurrency control mechanism since the primary site strategy can already handle partition failures. (See Davidson [1982] for a discussion of the relationship between the Optimistic Protocol and common forms of concurrency control.) It is also important to consider

the performance of proposed strategies, although it is difficult to obtain informed estimates on performance trade-offs. In some cases this results from the fact that an appropriate model is difficult to construct; in others it results from the fact that the mechanism is highly application dependent.

With these cautions in mind, we group the factors that influence the choice of a strategy into three areas:

Environment. Included here are the properties of the network and the nature of the partitionings. An important consideration is whether partitionings are caused by failures or are the result of anticipated events. In the latter case, complete information about the characteristics of the partitioning, including duration and network topology, may be known, and this can be exploited in some strategies (in particular, class conflict analysis).

However, most systems partition because of failures, and in this case the robustness of the strategy may be an important factor. For example, a primary site strategy would be a poor choice if site failures cannot be distinguished from communication failures. Also, class conflict analysis (as presented) cannot be used if communication failures do not always result in clearly delineated partitions.

The duration of the partitioning is also important. Long failures tend to generate many conflicts between transactions in different partitions; in this case, a pessimistic strategy is a better choice than an optimistic one.

Work Load. Two important work-load characteristics are average transaction length and transaction variance. Optimistic policies work better when transactions are short and variance small.

Another important work-load factor is locality of reference: Do updates to given data items tend to occur at a certain site? If so, a primary site strategy will not prohibit many transactions and availability will still be good. The rollback rate in the Optimistic Protocol will also be reduced, but the transactions will still have to be tested for conflict.

Application Specificity. These factors fall into two groups. The first are requirements placed by the application on transaction processing. Two important questions are

(1) Can transaction processing be temporarily halted for recovery purposes? If not, a pessimistic approach should be adopted which merely requires the forwarding of updates to merge the databases.
(2) Can transaction processing be limited in parts of the database, or is availability a premium? If the latter is the case, a more optimistic approach should be used.

The second group includes semantic considerations. Relevant questions here are

(1) Can transactions be rolled back? That is, do they have an inverse? If the latter is the case, either conflict should be avoided totally, or the divergent databases should be patched up by using compensating actions if necessary to achieve correctness.
(2) Is serializability a concern, or is a more procedural definition of "correctness" in the final database state acceptable? If serializability is not a major concern, a Data-Patch approach can be used.
(3) Should a partitioned system be expected to behave exactly as an unpartitioned system? For example, even if serializability is the "normal" correctness criterion, under extenuating circumstances (such as partition failures) a more lenient definition could be used.

6.2 Future Directions

Partitioned operation is still very much an active research area. We comment briefly on several interesting research directions.

One obvious deficiency in our current knowledge of partition strategies is the lack of any performance data on how well they work. Few strategies have been implemented and none tested on a representative application. Clearly, more experience with the proposed strategies is needed before we can understand the performance trade-offs between them.

Another important area of research is the adaptation of these strategies to accommodate more general processing models, in particular, nested transactions (and the related concept of multilevel atomicity [Garcia 1983; Lynch 1983]). Nested transactions arise in general purpose distributed programming environments such as ARGUS [Liskov and Schleifer 1983].

Algorithms for detecting and analyzing network partitions have also not been developed. Since several of the strategies require that the failure initially be recognized, this is an important area to address.

ACKNOWLEDGMENTS

This material is based on work partially supported by the National Science Foundation under grant ECS-8303146, and a fellowship from the IBM Corporation.

The authors are grateful to the referees and technical editor for suggesting numerous technical and stylistic improvements to this paper.

REFERENCES

ALSBERG, P. A., AND DAY, J. D. 1976. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, Oct.). IEEE Computer Society, Long Beach, Calif., pp. 627–644.

APERS, P. M., AND WIEDERHOLD, G. 1984. Transaction classification to survive a network partition. Unpublished manuscript, Computer Science Dept., Stanford Univ., Stanford, Calif. (July).

BERNSTEIN, P. A., AND GOODMAN, N. 1980. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th Internationl Conference on Very Large Data Bases* (Cannes, France, Sept. 9–11). IEEE, New York, pp. 285–300.

BERNSTEIN, P. A., AND GOODMAN, N. 1981. Con currency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June), 185–221.

BERNSTEIN, P. A., AND GOODMAN, N. 1983. The failure and recovery problem for replicated databases. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Aug.). ACM, New York, pp. 114–122.

BERNSTEIN, P. A., SHIPMAN, D. W., AND ROTHNIE, J. B. 1980. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar.), 18–51.

BLAUSTEIN, B. T. 1981. Enforcing database assertions: Techniques and applications. TR-21-81, Aiken Computation Laboratory, Harvard Univ., Cambridge, Mass.

BLAUSTEIN, B. T., GARCIA, H., RIES, D. R., CHILENSKAS, R. M., AND KAUFMAN, C. W. 1983. Maintaining replicated databases even in the presence of network partitions. In *Proceedings of the IEEE 16th Electrical and Aerospace Systems Conference* (Washington, D.C., Sept.). IEEE, New York, pp. 353–360.

DAVIDSON, S. B. 1982. An optimistic protocol for partitioned distributed database systems. Doctoral dissertation, Dept. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, N.J. (Oct.).

DAVIDSON, S. B. 1984. Optimism and consistency in partitioned distributed database systems. *ACM Trans. Database Syst. 9*, 3 (Sept.), 456–481.

EAGER, D. L., AND SEVCIK, K. C. 1983. Achieving robustness in distributed database systems. *ACM Trans. Database Syst. 8*, 3 (Sept.), 354–381.

EL ABBADI, A., SKEEN, D., AND CRISTIAN, F. 1985. An efficient, fault-tolerant algorithm for replicated data management. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems* (Portland, Ore., Mar.). ACM, New York, pp. 215–229.

ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov.), 624–633.

FISCHER, M. J., AND MICHAEL, A. 1982. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (May). ACM, New York, pp. 70–75.

GARCIA, H. 1982. Elections in a distributed computing system. *IEEE Trans. Comput. C-31*, 1 (Jan.), 48–59.

GARCIA, H. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst. 8*, 2 (June), 186–213.

GARCIA, H., AND WEIDERHOLD, G. 1982. Read-only transactions in a distributed database. *ACM Trans. Database Syst. 7*, 2 (June), 209–234.

GARCIA, H., ALLEN, T., BLAUSTEIN, B., CHILENSKAS, R. M., AND RIES, D. R. 1983. Data-Patch: Integrating inconsistent copies of a database after a partition. In *Proceedings of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems* (Oct.). IEEE, New York, pp. 38–48.

GIFFORD, D. K. 1979. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec.). ACM, New York, pp. 150–162.

GIFFORD, D. K., AND SPECTOR, A. 1984. The TWA reservation system. *Commun. ACM 27*, 7 (July), 650–665.

GOODMAN, N., SKEEN, D., CHAN, A., DAYAL, U., FOX, S., AND RIES, D. 1983. A recovery algorithm for

a distributed database system. In *Proceedings of the 2nd ACM Symposium on Principles of Database Systems* (Atlanta, Ga., Mar.). ACM, New York, pp. 8–15.

GRAY, J. N. 1978. Notes on data base operating systems. IBM Res. Rep. RJ2188 (Feb.). Also published in *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmuller, Eds. Springer-Verlag, Berlin and New York, 1979, pp. 393–481.

GRAY, J. N., LORIE, R. A., PUTZOLU, G. R., AND TRAIGER, I. L. 1976. Granularity of locks and degrees of consistency in a shared database. In *Modeling in Database Management Systems*, G. M. Nijisen, Ed. Elsevier North-Holland, New York, pp. 365–394.

GRAY, J. N., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. 1981. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2 (June), 223–242.

HAMMER, M. M., AND SHIPMAN, D. W. 1980. The reliability mechanisms of SDD-1: A system for distributed databases. Computer Corporation of America Tech. Rep. CCA-80-04 (Jan.). (A shorter version appears in *ACM Trans. Database Syst. 5*, 4 (Dec.), 431–466.)

HERLIHY, M. P. 1984. General quorum consensus: A replication method for abstract data types. Tech. Rep. CMU-CS-84-164, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa. (Dec.).

HERLIHY, M. P. 1985. Using type information to enhance the availability of partitioned data. Unpublished manuscript, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa. (Apr.).

KOHLER, W. 1981. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv. 13*, 2 (June), 149–184.

KUNG, H. T., AND ROBINSON, J. T. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June), 213–226.

LAMPORT, L. 1978. Time, clocks and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LIPSKI, W. 1979. On semantic issues connected with incomplete information databases. *ACM Trans. Database Syst. 4*, 3 (Sept.), 262–296.

LISKOV, B., AND SCHEIFLER, R. 1983. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July), 381–404.

LYNCH, N. A. 1983. Multilevel atomicity—A new correctness criterion for distributed databases. *ACM Trans. Database Syst. 8*, 4 (Dec.), 484–502.

MINOURA, T., AND WIEDERHOLD, G. 1982. Resilient extended true-copy token scheme for a distributed database system. *IEEE Trans. Softw. Eng. SE-8*, 3 (May), 173–189.

MOHAN, C. 1980. Distributed data base management: Some thoughts and analyses. In *Proceedings, ACM Annual Conference* (Nashville, Tenn., Oct.). ACM, New York, pp. 399–410.

PAPADIMITRIOU, C. H. 1979. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct.), 631–653.

PARKER, D. S., AND RAMOS, R. A. 1982. A distributed file system architecture supporting high availability. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Pacific Grove, Calif., Feb.). Lawrence Berkeley Laboratory, University of California, Berkeley, Calif., pp. 161–183.

PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B., WALTON, E., CHOW, J., EDWARDS, D., KISER, S., AND KLINE, C. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng. 9*, 3 (May).

POPEK, G., WALKER, B., CHOW, J., EDWARDS, D., KLINE, C., RUDISIN, G., AND THIEL, G. 1981. Locus: A network transparent, high reliability distributed system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec.). ACM, New York, pp. 169–177.

ROTHNIE, J. B., AND GOODMAN, N. 1977. A survey of research and development in distributed database management. In *Proceedings of the 3rd International Conference on Very Large Data Bases* (Tokyo, Japan, Oct. 6–8). IEEE, New York, pp. 48–61.

SARIN, S., BLAUSTEIN, B., AND KAUFMAN, C. 1985. System architecture for partition-tolerant distributed databases. *IEEE Trans. Comput. C-34*, 12 (Dec.), 1158–1163.

SKEEN, D. 1982a. A quorum-based commit protocol. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Pacific Grove, Calif., Feb.). Lawrence Berkeley Laboratory, Univ. of California, Berkeley, Calif. pp. 69–80.

SKEEN, D. 1982b. Crash recovery in a distributed database system. Doctoral dissertation and ERL Memo M82/45, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley (May).

SKEEN, D. 1982c. On network partitioning. In *Proceedings of the IEEE Computer Software and Applications Conference (COMPSAC)* (Nov.). IEEE, New York, pp. 454–455.

SKEEN, D., AND STONEBRAKER, M. 1983. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng. SE-9*, 3 (May), 219–228.

SKEEN, D., AND WRIGHT, D. 1984. Increasing availability in partitioned networks. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Apr.). ACM, New York, pp. 290–299.

STONEBRAKER, M. 1979. Concurrency control and consistency of multiple copies in distributed INGRES. *IEEE Trans. Softw. Eng. SE-5*, 3 (May), 188–194.

THOMAS, R. H. 1979. A majority consensus approach to concurrency control. *ACM Trans. Database Syst. 4*, 2 (June), 180–209.

TRAIGER, I. L., GRAY, J. N., GALTIERI, C. A., AND LINDSAY, B. G. *1982.* Transactions and consistency in distributed database systems. *ACM Trans. Database Syst. 7*, 3 (Sept.), 323–342.

WRIGHT, D. D. 1983. Managing distributed databases in partitioned networks. TR83-572, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y. (Sept.).

Received March 1985; revised October 1985; final revision accepted December 1985.