RELIABLE DISTRIBUTED DATABASE MANAGEMENT

Hector Garcia-Molina

Robert K. Abbott

CS-TR-047-86

August, 1986

Reliable Distributed Database Management

Hector Garcia-Molina
Robert K. Abbott

Department of Computer Science
Princeton University
August 1986

A reliable distributed database must manage dispersed and replicated data, making it available to users in spite of hardware failures. In this paper we study the algorithms and techniques that can achieve this reliability. Three scenarios are considered. In the first, data are distributed but not replicated; in the second, data are replicated; and in the third, both data and processing are fully replicated in an n-modular redundancy strategy. Network partitions are not considered in any scenario.

Keywords: distributed database, transaction, replication, reliability, availability, consistency, concurrency control, serializability, atomicity, commit protocol, crash recovery, deadlock, fail-stop, fail-insane, n-modular redundancy, Byzantine Agreement

This work has been supported by NSF Grants ECS-8303146, DMC-8505194, and ECS-8351616, New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from DEC, IBM, NCR, and Perkin-Elmer corporations.

A reliable database management system should provide its users with correct data whenever they need it. Reliability can be achieved through distribution and replication of resources (e.g., data, processors); distribution isolates failures and replication makes alternate resources available. However, in order to actually achieve reliability, the distributed and replicated resources must be properly managed by the system. In this paper we study such management.

Our goal is to explain the basic principles of reliable, distributed data management, rather than to exhaustively survey the field. Thus, in each section of the paper, we usually select one of the many possible approaches or techniques because it is, in our opinion, either simpler to understand or more commonly used in practice. Interestingly, the simple ideas are the ones that usually work best in practice. This is especially true in reliable data management, where simpler means less prone to errors and hence more reliable. Our presentation is informal since we believe that once the reader understands the basic ideas, he or she can refer to the literature for details and proofs.

Since data replication is one of the keys to reliability, and affects in fundamental ways the type of crash recovery that can be performed, our paper is organized by increasing levels of replication. In section 1, we introduce the terms and basic concepts of database management, including our definitions of reliability and failures. Section 2 considers data management in a distributed system with no data replication. It lays the groundwork for section 3, which discusses a system with replicated data. Finally, section 4 covers a system where both data and computations are replicated.

1. Basic Concepts

A distributed system is a collection of autonomous computing nodes linked by a communications network. The nodes may cooperate to provide the user a database management system (DBMS) in which case the distributed system is called a distributed database. There is no requirement that nodes be homogeneous either in terms of hardware, or in the data models used, or in the database management systems. In reality, however, management of heterogeneous databases is very difficult [CP84].

A distributed database is a collection of named data items, each with an associated value. In an *unreplicated* distributed database each item has exactly one value. In a *fully replicated* distributed database an item has **n** associated values where **n** is the number of nodes in the system. Each value for a given item is stored at a different node in the system. The values of the item should be the same, but due to updating activity the values may be temporarily different. The values stored at a node constitute the database of that node. A distributed database which is neither unreplicated nor completely replicated is called a *partially replicated* distributed database.

In addition to items and values, a distributed database has a collection of consistency constraints [EGLT76]. A *consistency constraint* is a predicate defined on the distributed database describing the relationships that must hold among the items and their values. A consistency constraint defined on a database for a bank might be that all balances must be non-negative.

Two operations that can be performed on data items are the read action and the write action. A *read* action retrieves the current value of an item and a *write* action updates an item with a new value. Operations on the data are grouped into a *transaction,* a program that issues read and write actions. The items read by a transaction constitute its *readset*; the items written, its *writeset.* A transaction may also have effects external to the database, such as dispensing money or printing a report about the database. A *read-only* transaction issues neither write requests nor has external effects.

Two transactions *conflict* if and only if they operate on a common data item and at least one of the operations is a write. If one reads and the other writes, it is a *read-write conflict*; if they both write, a *write-write* conflict. The order of execution of the actions of the two transactions is significant (i.e., reflected in the database state) only if they conflict. Consider a database system that is used to maintain savings account balances for a bank. Suppose there are three accounts **A**, **B** and **C** which contain initially $50, $100 and $150 respectively. Let **T1** be a transaction that transfers $10 from account **A** to account **B** , and let **T2** be a transaction that transfers $50 from account **B** to account **C**. The situation is illustrated in figure 1 [Kohl81].

```
Initial account balances
   A contains $50
   B contains $100
   C contains $150

Transaction T1
   begin
   1    read A obtaining A_balance
   2    read B obtaining B_balance
   3    write A_balance – $10 to A
   4    write B_balance + $10 to B
   end

Transaction T2
   begin
   1    read B obtaining B_balance
   2    read C obtaining C_balance
   3    write B_balance – $50 to B
   4    write C_balance + $50 to C
   end
```

Figure 1. Conflicting transactions.

Since both transactions are funds transfers, the sum of the balances after the transactions have run should equal the sum of the initial balances, namely $300. If **T2** is scheduled to run between the first and second write actions of **T1** then the resulting sum of the account balances will be $350. This anomaly is called a lost update (the deduction of $50 from account **B** by **T2** is lost) and is only one example of anomalies that can arise from conflicting transactions [Gray79]. Conflicts must be resolved correctly if consistency is to be preserved.

One solution for resolving conflicts would be to execute transactions serially, one at a time in any order. This method preserves consistency because a transaction, when executed alone, *transforms an initially consistent database state into another consistent state* [TGGL82]. However this method prohibits the concurrent execution of nonconflicting transactions and severely reduces system performance.

A better solution is to execute transactions concurrently, but ensuring that from the users' viewpoint, they appear as indivisible operations on the database. This property is called *atomic execution* (also concurrency transparency) and is achieved by guaranteeing the following two properties:

1. A transaction is an "all or nothing" operation. Either all its actions are executed and the effects properly installed in the database, in which case the transaction is committed, or all effects of the transaction are undone and the transaction is aborted. This property is known as *atomic commitment.*
2. The concurrent execution of several transactions affects the database as if they executed serially in some order. The interleaved order of the actions of a set of concurrent transactions is called a *schedule.* A schedule is *serializable* if the effect of running the schedule is the same as if the transactions had executed serially in some order.

The first property is established by the commit and recovery algorithms of the database management system; the second, by the concurrency control algorithm. Atomic transaction execution together with the consistency-preserving assumption imply that the execution of any set of transactions transforms a consistent initial database state into a consistent state. Atomicity is appealing because of its simplicity and generality. It also corresponds to most users' intuitive model of processing - that of sequential processing.

1.1 Reliability

An important component of any computing system is reliability. In fact, a major motivation for building distributed database systems is to increase reliability [Kim84]. When the computer for a centralized database fails, the entire database is unavailable until the computer is repaired. When a node fails in a distributed system, only a portion of the database is made unavailable. If that portion is replicated at another node, then the entire database may remain available.

Intuitively, reliability is a measure of how well a system can tolerate and recover from failures. (In this discussion we specifically mean hardware failures. All software is assumed to work correctly.) Our discussion will focus on two aspects of reliability: *correctness* and *availability.* Correctness is important because we want the system to maintain database consistency. Failures may temporarily cause an inconsistent state but the recovery algorithms should restore the database to a correct state. Issues related to correctness are serializability, atomicity and survivability. Updates to the database should survive, i.e., failures should not cause updates to be "lost".

The second aspect of reliability is availability. Failures will make portions of the database unavailable. However, by replicating data at independent computing nodes we can minimize (or even nullify) the impact of node failures. The goal is to design a system that can sustain multiple node failures and continue to process transactions promptly and correctly.

Reliability does not come for free. To some extent, reliability is achieved through redundancy, and redundancy naturally increases the cost of the system. Intelligent design decisions will seek to minimize the cost and maximize efficiency. A thorough examination of the issues of cost, efficiency and fairness are beyond the scope of this paper.

1.2 Single site reliable database

In this section we outline some of the mechanisms used to achieve reliability for a single site database. Specifically we discuss concurrency control in section 1.2.1 and crash recovery in section 1.2.2. It is by no means a comprehensive treatment of the subject. Rather the purpose is to familiarize the reader with some of the problems and terms of concurrency control and crash recovery. When we consider distributed databases, we will assume that individual nodes have the capabilities of a centralized database system.

1.2.1 Concurrency Control

Most commercial database management systems solve the concurrency control problem by some form of explicit or implicit *locking* [Gray79, BG81]. A transaction locks an item to make it inaccessible to other transactions while the database is in a temporarily inconsistent state. A simple locking scheme uses two kinds of locks: *read locks* and *write locks*. A transaction uses a read lock if it just reads an item and a write lock if it updates an item. Read locks are sharable; more than one transaction can lock an item for reading at the same time. Write locks are owned exclusively and are incompatible with any other type of lock.

In order to guarantee serializability all transactions must be *well-formed* and *two-phase*. A transaction is well-formed if it

1. locks an item before accessing it,

2. does not lock an item in a way that is incompatible with existing locks,

3. before it completes, unlocks each item that it locked.

A transaction is two-phase if no item is locked after some item has been unlocked.

If a transaction attempts to lock an item in a way that is incompatible with existing locks then it must either wait, abort itself or preempt the other transaction. When two or more transactions are waiting, they may be involved in a *deadlock*. Figure 2 is a simple example of a deadlock; each transaction is waiting to lock an item that is already locked by another transaction. Deadlocks are

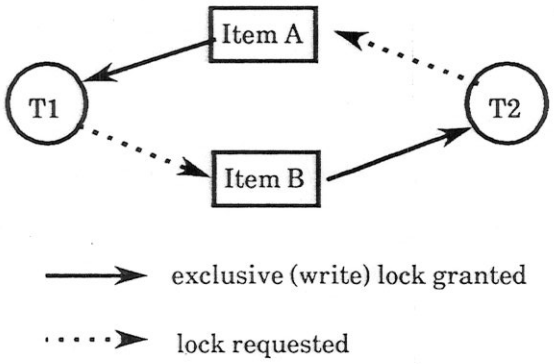

Figure 2. Deadlocked transactions.

resolved by preempting one or more of the transactions that are involved. Methods for representing, avoiding and detecting deadlocks are reviewed in [MM79, Ober82].

Two-phase locking is perhaps the best-known concurrency control mechanism but it is not the only one. Concurrency control is a well-studied problem and many solutions have been proposed [Ullm82].

1.2.2 Crash Recovery

A failure in a computing system is an undesired event which leaves the system in an inconsistent or erroneous state. A failure can be relatively minor (e.g., transaction abort, deadlock, timeout, or resource limit exceeded) or major (e.g., multiple node failures). *Recovery management* is the process by which the system returns to a correct state after a failure. A *recovery technique* is a mechanism that is applied to recover from a certain class of failures. There are many kinds of failure and therefore many different recovery techniques.

The purpose of a recovery system is twofold [Gray79]. First, the recovery system provides a way to undo an in-progress transaction in the event of a minor error without affecting other transactions. Second, in the event of a major error, the recovery system attempts to minimize the amount of lost work as it restores the database to a previous correct state. We will describe one approach to recovery, namely logging. Many other mechanisms for crash recovery have been proposed [Verh78, HR83]. Deciding which method is best for a particular database system depends on how many, and what kinds of transactions are being run [Kent84].

A transaction starts with a **BEGIN** action which establishes a recovery unit. Then it issues a sequence of read and write actions on the database items, and terminates with an **END** action that marks the end of the recovery unit. Once the last action is executed, the system *commits* the transaction by making all changes to the database permanent, releasing all locks held by the transaction, and making the changes public. The *commit point* occurs when the system makes the decision to commit the transaction. At any time before the commit point the transaction can be *aborted.* If a transaction is aborted, all the effects of the actions are undone and the database is restored to its original values. This is known as *transaction rollback.* (Incidentally, note that in addition to the two-phase locking requirement, locks on modified items should be held until after the commit point.) Commitment and rollback are accomplished with the information stored in the **UNDO-REDO** log (or audit trail) [Gray79]. Every action that updates an item in the database writes in the log an entry that includes the old and the new values of the item being changed. The information should be sufficient to enable an action to be completely undone or redone. Since entries for all transactions are written to a common log, each entry must include a transaction identifier. For reliability, several copies of the log may be written at once. In addition, log entries are also written when the transaction begins and when it reaches its commit point. A log also has the following associated operations [Gray79]:

1. A **DO** operation that performs the action and also writes a log entry sufficient to undo and redo the action.

2. An **UNDO** operation that undoes the action given the log entry written by the **DO** operation.

3. A **REDO** operation that redoes the action given the log entry written by the **DO** operation.

If an update is made to the permanent database and a crash occurs before the relevant log entry is written to stable storage, then it is impossible to undo the update. The solution is to write the log to stable storage before the update is done. This is known as the *write-ahead log protocol* [Gray79].

In the event of a crash, the log is used to reconstruct a consistent and up-to-date copy of the database. If the database was not destroyed in the crash, the recovery system will redo (i.e., complete) transactions that were committed but whose actions on the database had not fully completed (e.g., the system may have not flushed out all updates to the database on disk). Similarly, the system will use the log to undo transactions that had not committed but whose changes to the database could have been executed. If the database is lost in the crash, then the system relies on the latest database dump (a consistent but out-of-date copy of the database) and redoes all transactions that committed after the time of the dump. The operations **UNDO** and **REDO** must be *idempotent*, meaning that repeated application on the data yields the same desired result. This is necessary because a node may experience multiple failures during the course of transaction recovery.

Not every action of a transaction can be undone. For example, once an automatic teller has dispensed cash, the money cannot be recovered. We call those actions that can be undone *recoverable actions*. All other actions are *unrecoverable*.

1.3 Failures in a Distributed Environment

In order to design recovery mechanisms we first need to identify what kinds of failures the system will protect against. Let's look at the types of failures that could occur in a distributed database system.

- Processor failure. There are a number of ways in which a processor can cease to function normally. It might simply halt. Or it could continue to execute instructions correctly but with intermittent and abnormal delays. Finally, a processor can go "berserk" and exhibit wild, unpredictable behavior. Systems that protect against this type of insanity are very reliable indeed.
- Storage failures. A computing system usually employs several types of storage media each with its own failure characteristics. Nevertheless we can divide storage media into two broad classes. The

contents of *volatile* storage (e.g., main memory) do not survive processor failures. *Nonvolatile* storage (e.g., magnetic disks and tapes) does. Storage devices can also experience failures which are unrelated to processor failures.

• Network failures. These occur when messages between nodes are not transmitted properly. Let us assume that the protocol for sending a message from node **X** to node **Y** is as follows: node **X** hands the message to the communications subsystem at time **t**, and **d** seconds later **X** gets a response from the communications subsystem. (Here we use time in an intuitive fashion but these concepts can be formalized [Lamp78].) This response can be either positive ('message has been delivered') or negative ('message has not been delivered'). The delay **d** should be less than some predefined maximum acceptable delay $\mathbf{d_m}$. At node **Y** there are several possible outcomes: the message may never be delivered; an erroneous message due to transmission errors may be delivered; the correct message may be delivered before $\mathbf{t+d}$ but it may be out of sequence with respect to other **X - Y** messages; the correct message may be delivered in sequence but after time $\mathbf{t+d}$, and so on. This means that there are a large number of combined outcomes (e.g., no response at node **X** by time $\mathbf{t+d_m}$ but message is delivered at node **Y** in sequence before time $\mathbf{t+d_m}$).

• Partitions. Node failures or network failures can partition the distributed database into groups of isolated nodes. The nodes in each group can communicate with each other, but no node in the group is able to communicate with nodes in other groups. Whenever partitions occur the correctness of the database is threatened. For example, suppose the distributed database of a bank is partitioned into two isolated groups of nodes **P1** and **P2**. Before the partitioning occurred the Silicon Co. showed a balance of $150. It is obviously incorrect to allow both a node in **P1** and a node in **P2** to submit a transaction to withdraw $100 from the company account, even though each partition thinks there are sufficient funds to allow the transaction.

• Malevolent failures. A malevolent failure occurs when a node acts maliciously and does not follow the system protocols. An example would be if a node started generating and processing transactions that do not preserve consistency. Clearly, it is very hard to protect against such failures.

• Multiple failures. A multiple failure means that more than one of the failures discussed so far has occurred. If a transaction processing algorithm can still function properly after **n** failures, it is called *n-resilient.* Algorithms that recover from multiple failures must guard against the situation where a failure occurs when the system is recovering from a previous failure.

• Detectable failures. If all nodes directly affected by a failure can recognize or discover that a failure has occurred, the failure is called *detectable.* But if the nodes continue to operate as if no failure occurred, the failure is called *undetected.* For example, if a node's database is destroyed (e.g., by disk failure) and the node does not immediately recognize this, the hard node failure is called undetected. Clearly, this undetected failure can cause serious problems, as the failed node may attempt to modify other nodes' data based on its own erroneous data. Similarly, an undetected failure occurs when a node accepts as good a message that was altered by transmission errors.

For another discussion of possible failures and their problems, see [RG77].

No system can protect, nor would we want to protect, against all types of failures. Some (e.g., flooding) may be unlikely to occur in a given installation. Others (e.g., an earthquake) may be deemed too expensive to handle. Of the failures we would like to cover, some are easy to handle (e.g., out-of-sequence messages); for others there are well-known techniques that make processor failures "invisible" to transactions running at a single computer (e.g., logging, as discussed in section 1.2.2). To avoid reinventing known solutions, and to focus system design on precisely the critical failures, it is common to make assumptions about the components. We make ours in the following section.

1.4 Modeling Failures

Following [LS79], we divide all events which affect the computing system into two categories: *desired* and *undesired.* Understandably, desired events are what we always hope for. Any kind of failure is an undesired event. Undesired events can be further classified as *expected* or *unexpected.* A processor failure or an undelivered message are typically undesired, but expected events. An earthquake is usually both undesired and unexpected.

We next consider two components: a node (processor plus storage) and the communication network, and present the most common failure models for each. Each model makes different

assumptions about the reliability of the components. We consider node failure models first and the network model second.

1.4.1 Node Models

A node can be in one of four states. In the *perfect* state, a node experiences no failures. It follows the algorithms it is given, and never pauses or halts. Furthermore, the algorithms it follows are correct. In the perfect state a node responds promptly to messages from other nodes.

When a node is in the *insane* state, it can act in arbitrary ways. It can send any messages, including garbled or misleading ones, to other nodes. It can refuse to send messages when it is supposed to. It can even collaborate with other failed nodes in an attempt to subvert the entire system. The node can also destroy all the data that it holds.

During *recovery*, a node executes a predefined set of algorithms which restore the database to a consistent state. The node is perfect in the sense that it executes algorithms faithfully. However, the node does not execute transactions because the database state is incorrect.

Finally, when a node is *halted*, it performs absolutely no actions.

Based on these four states, we now define two node models that tell us how transitions between the states can occur.

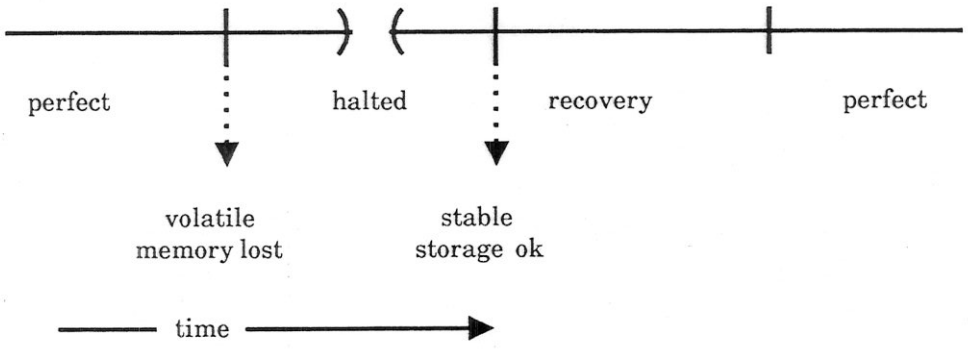

Figure 3. Fail-stop node model.

A node is defined to be *fail-stop* if it simply halts when it fails [SS83]. Figure 3 illustrates the behavior of such a node. Before a failure, the node runs perfectly. The moment it fails, it simply halts, without ever deviating from its algorithms. After some undetermined amount of time, the node is repaired. When this happens it executes a predefined recovery procedure. When recovery is complete, the node continues executing its application programs perfectly. (The node can fail during recovery, in which case it again halts.) A fail-stop node has two types of storage. Main memory is volatile and is lost at a crash. Stable storage is nonvolatile and updates to it are performed atomically. That is, if a failure occurs during a write to nonvolatile storage, either the write will be performed entirely or none of it will. There are a number of techniques for increasing the probability that a real node behaves like a fail-stop one [SS83], but of course this probability will never be 1. For example, the logging mechanism of section 1.2.2 can be helpful in making storage (i.e., the database) behave like stable storage.

A *fail-insane* node operates perfectly until it fails (see node A in figure 4). During the failure it is in the insane state. At some later point, the failure is detected and repaired. After a recovery period, the node behaves perfectly again. (The node may fail insanely during recovery too.)

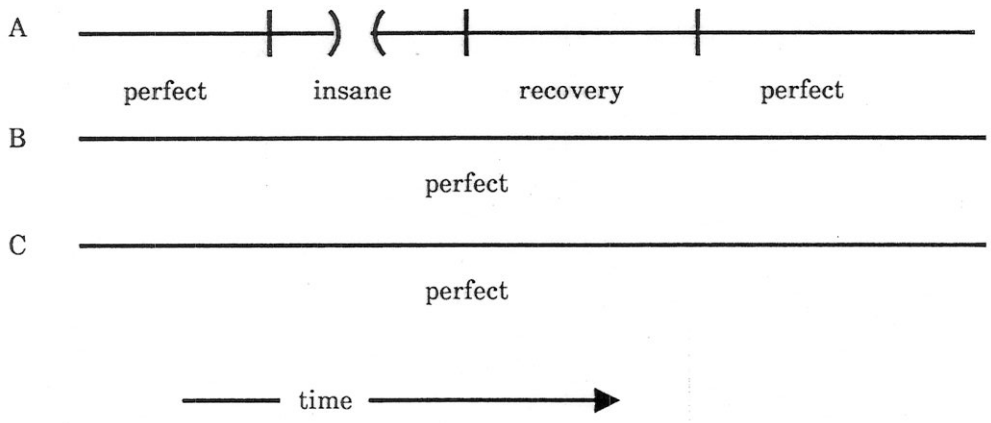

Figure 4. Fail-insane node model

A fail-insane node by itself is not very useful for reliable computing. Thus, we assume that there are $2m+1$ fail-insane nodes in the system, and that at any one time, at most m are insane or recovering. This way, the majority of perfect nodes can perform useful work and rule out any incorrect or misleading results produced by the insane nodes.

More details on the fail-insane model will be given in section 4, but note that it is a very powerful model. It covers any conceivable failure, regardless of whether it was considered by the system designers. It is the most conservative model, and any system that protects against this type of failure will be highly reliable.

1.4.2 Network Model

Computer communications networks are relatively well-understood today, and various techniques for making them robust are known. It is also possible to model some communications failures (e.g., lost messages) as failures of either the sending or receiving node. Thus, from the distributed database point of view, it is common to model the network as a very reliable one. Specifically, we make the following assumptions for a *reliable network*:

- Network does not lose messages. We assume that all messages will be delivered unless the destination node is down.
- Inorder messages. All messages are delivered in the order in which they were sent.
- Guaranteed delivery time. We assume that a sending node will always get some response from the communication network after at most d_m seconds, and that a positive response means the message has been delivered. A negative response from the communications subsystem means that the receiving node is failed.
- No spontaneous messages. The network only delivers authentic messages sent by nodes in the system. It does not spontaneously generate and send its own messages.
- No network partitions. The system does not split into isolated groups of nodes which cannot communicate with each other.

Note that in this model there are no "persistent messages" (as in RELNET [HS80]). That is, if the destination node is down, the network will not remember messages intended for it. The source

node must take this responsibility. (We choose not to assume persistent messages because their implementation is not simple. Furthermore, many distributed database systems will have logging facilities that make persistent messages unnecessary.)

A second model that has been studied is the *partitionable network*. In it, network partitions are considered a possibility. Specifically, the guaranteed delivery time assumption of the previous model is relaxed. If a node is unable to send a message to another node, that other node may either be failed or may simply be cut off. Note that network partitions can lead to some difficult data management problems. For instance, if isolated groups of nodes update replicated data during a partition, then the "copies" will diverge and will have to be integrated when the partition is repaired.

To limit the length of this paper, we will not study the partitionable network model. Interested readers are referred to [DGS84] for a discussion of reliable distributed data management under network partitions. Hence, we will study in a reliable network environment, data processing with fail-stop nodes (sections 2 and 3) and with fail-insane nodes (section 4).

2. No Data Replication

In this section we discuss how to achieve reliability in an unreplicated distributed database. Recall that in an unreplicated database there exists exactly one copy of each data item.

The model for the distributed system is a collection of fail-stop computing nodes connected by a reliable, non-partitionable communication network. An unreplicated database cannot tolerate fail-insane nodes. A fail-insane node can arbitrarily destroy any part of the database and, since there exist no duplicate copies for any data items, the damaged items cannot be reconstructed.

Non-replication simplifies transaction processing because there is no need to propagate updates to multiple copies. Conflict avoidance among transactions is also simplified. If item **X** is part of the local database of node **P**, then access to **X** can be controlled exclusively by **P**. However, non-replication also means that the availability of the data is limited. If node **P** is failed then all transactions which reference **X** must wait until **P** is repaired.

Because data availability is limited, we will concentrate on the mechanisms used to maintain correctness. Specifically these are the concurrency control mechanism, (discussed for the single node case in section 2.1), and the commit protocol (section 2.2), which are used to achieve atomicity. To be reliable these algorithms must operate correctly in the face of multiple node failures. Likewise, the associated recovery algorithms must also be immune to failures.

We model a distributed transaction as a collection of processes, or sub-transactions (often called cohorts) which execute at various nodes in the network. An instance of a transaction is created by the arrival of a transaction request message which initiates the construction of a process from a transaction descriptor. The *transaction descriptor* provides complete instructions for building an instance; it indicates what sub-transactions are needed, how to schedule the transaction, what recovery and locking mechanisms should be used, and so on [Kohl81].

The simplest way to organize the sub-transactions is sequentially. In this model only one sub-transaction of a transaction executes at any time in the network. If the sub-transaction acquires all necessary resources and is able to successfully complete its portion of the transaction, it spawns the next sub-transaction to be executed. The entire transaction completes successfully only if the last sub-

transaction in the sequence completes. Otherwise each sub-transaction in the sequence is rolled back. This means that no sub-transaction can make its database updates available to other transactions until it is determined that the entire transaction has been successful. A sub-transaction that executes at the same node as its parent can access the same objects as its parent. In this way a sub-transaction can "see" the actions of its predecessor.

Rothnie et al. have proposed another model for a transaction [Roth80]. In this model the sub-transactions are spawned and managed by a single process known as the transaction coordinator. The transaction coordinator does not itself perform any database operations. This model does allow sub-transactions to execute concurrently.

Our model of a distributed transaction is more general. A transaction initiated at the start node spawns a set of sub-transactions. Similarly each sub-transaction can spawn its own set of cohorts. These relationships are shown in figure 5. Notice that concurrent execution of sub-

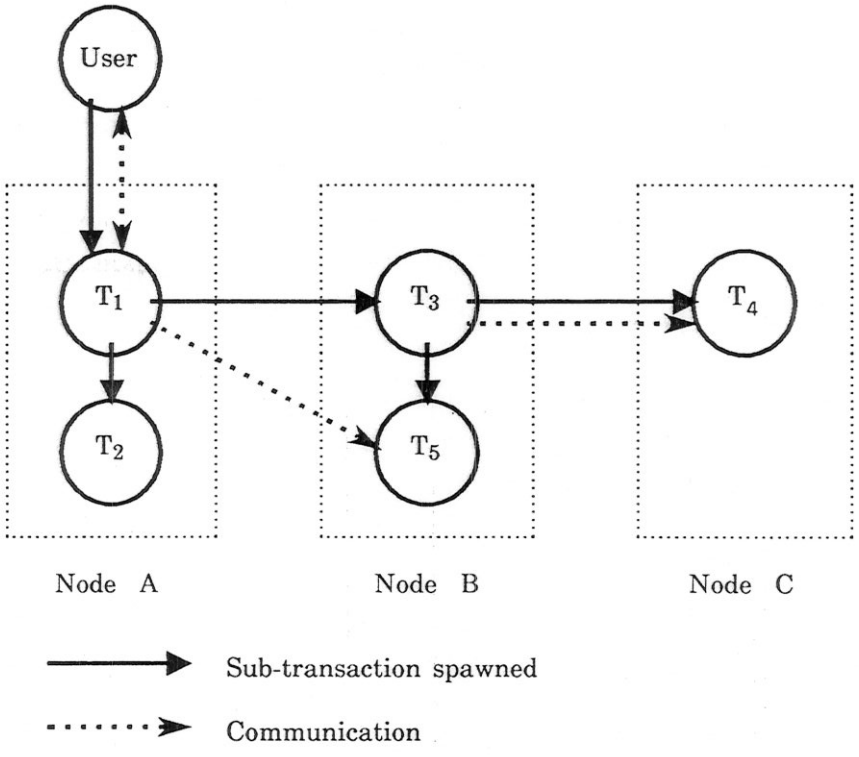

Figure 5. Distributed transaction model

transactions in the more general model enables sub-transactions at different nodes and of different parentage to share results through the explicit passing of messages. In this model a transaction completes successfully if all of the sub-transactions are successful. Coordinating the sub-transactions so that either all of them commit or none of them do is known as the distributed commit problem.

2.1 Concurrency Control

We have seen that the key to achieving database correctness is to execute transactions atomically. The first step towards this goal is to ensure that all executions are serializable. Serializability is attained by processing conflicting operations in certain relative orders. In addition, a good concurrency control mechanism for a distributed system should be resilient to node failure and incur modest computational and storage overhead. There are many concurrency control mechanisms for distributed database systems; [BG81] is one survey of this area. Our goal is not to survey but to illustrate how concurrency control is affected by failures. Thus we summarize the best-known distributed concurrency control mechanism, namely two-phase locking, and use it to discuss failure management.We encourage the reader to consult the original papers for more formal presentations of the problems and proofs that the techniques we describe are indeed correct.

In a centralized database two-phase locking is implemented by a lock manager that receives lock requests and lock releases and processes them according to the rules of two-phase locking (section 1.2.1). The basic way to implement two-phase locking in a distributed database is to distribute the lock managers along with the database; placing a lock manager for data item **X** at the node where **X** is stored. A transaction that updates **X** must first spawn a sub-transaction to execute at the node where **X** is stored. The sub-transaction requests an exclusive lock for **X** from the local lock manager. If the lock is granted then the sub-transaction proceeds to update **X**. Once the lock has been granted no other transaction can access **X** until the lock is released. Similarly, shared locks must be requested before reading item **X**, and this lock is sharable with other readers. Sub-transactions and the local lock managers follow the rules of two-phase locking to ensure that executions are serializable at each node. Two transactions conflict only if they spawn at least one pair of conflicting sub-transactions. Because data is not replicated the conflicting sub-transactions compete for locks at

the same node in the network. The lock manager for that node will resolve the conflict and serialize the execution of the conflicting sub-transactions. Thus the serializability of local executions is sufficient to guarantee the serializability of global executions.

Lock requests which cannot be granted immediately are placed in a waiting queue for the desired item. As in a centralized database, uncontrolled waiting can lead to deadlocks. In a distributed system the local lock managers cooperate to help detect global deadlocks. There are a number of techniques for handling deadlocks in a distributed system. As in a centralized system they are characterized by the cautious approach of deadlock prevention and that of deadlock detection. For a thorough discussion of deadlock avoidance and detection see [CP84, MM79, Ober81].

The mechanism described so far synchronizes conflicting operations when there are no failures. Failures, however, do occur and can disrupt normal transaction processing. While a node is halted, there can be no conflicts over the items stored there since there is no activity. However, at recovery, the lock table no longer exists. During normal processing the lock table, which records what locks have been granted to which sub-transactions, is usually kept in main memory. (The main reason for this is efficiency.) Thus when a failure occurs the lock table is lost. After a node is repaired, and before it can resume normal processing, the lock table must be reconstructed to reflect the state of the system as it was just before the failure. We now discuss one way that this can be accomplished.

When a node recovers, there exists a set of sub-transactions which were active when the failure occurred. (This set can be constructed by examining the log and identifying sub-transactions which have recorded a **BEGIN** action but have not entered an **END** action.) Before these pending sub-transactions can resume execution, the lock table which existed at the time of the crash must be reconstructed. Using the information in the log this task is easily accomplished. For each pending sub-transaction T_i, the recovery procedure will

1. If possible, commit T_i.

A node can commit a pending sub-transaction T_i if it can determine that the transaction to which T_i belonged reached its commit point. How a node can determine upon recovery when it is possible to commit (or abort) a pending sub-transaction is discussed in the next section. If T_i has committed, the

node must ensure that T_i's changes are safely stored in the database. Since T_i has finished all of its processing, it no longer needs any locks. Thus, the reconstructed lock table need not contain any entries for T_i.

2. Else if possible, abort T_i.

A node can abort a sub-transaction if it can determine that the transaction to which T_i belonged aborted (see section 2.2). In this case, the node aborts T_i; after the abort T_i holds no locks, so it will have no lock table entries.

3. If T_i can be neither committed nor aborted, then the node locates in the log the **BEGIN** entry for T_i. Reading forward in the log, the node sets an exclusive lock for each item modified by T_i. These locks are held until the fate of T_i can be determined. Note that shared locks for items read do not have to be set. This is because we are not attempting to restart the transaction and let it continue processing. If the transaction had managed to finish before the failure it will be committed; else it will be aborted.

When all pending sub-transactions have been reviewed and the above actions applied, then the lock table has been reconstructed, and it is safe to resume normal processing.

2.2 Commit Protocols

We have modeled a distributed transaction as a set of sub-transactions which execute concurrently at nodes in the network. For correctness we require that either all the sub-transactions complete successfully and commit their actions or that none of them do. Atomic commitment is the second step toward achieving atomic execution of transactions. This section discusses commit protocols which handle the distributed commit problem.

To facilitate the discussion of commit protocols we introduce a more formal model of execution for a sub-transaction. The model consists of a set of possible states which the sub-transaction can enter and a set of rules which clearly define state transitions. The states and transitions are shown in Figure 6. The fail-stop model and the logging mechanisms at each node ensure that the state transitions in the diagram are, in essence, atomic. Each transition is defined to occur when the appropriate log record is written to stable storage, something that is assumed to occur atomically (see

section 1.4.1). Thus, for example, the transition to state C (commit) occurs when the commit record is written.

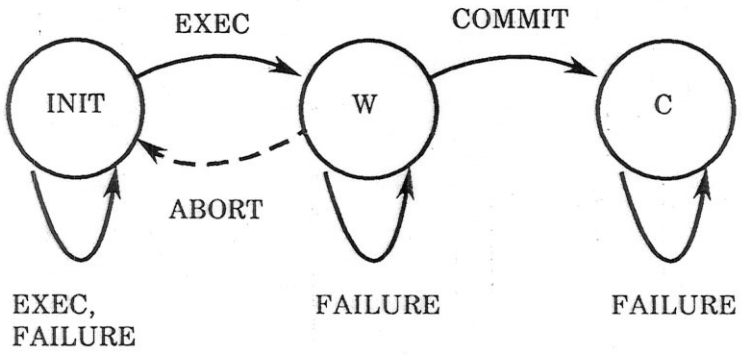

Figure 6. Sub-transaction model

Let the state of sub-transaction T_i before it begins execution at node P be called the initial state **INIT**. In the **INIT** state, T_i is a process waiting to be scheduled by the database system. From the **INIT** state T_i begins execution. It requests locks from the local lock manager and for every action, writes **UNDO** and **REDO** entries to the log. Execution has two possible outcomes: success and failure.

Failure can be caused by a concurrency control problem discovered by either the local lock manager or the global deadlock avoidance and detection algorithm. Using the log information, T_i is rolled back to the **INIT** state and scheduled for restart at a later time. Alternatively, T_i can decide to unilaterally abort. Using a banking example, T_i might discover that there are insufficient funds to cover a withdrawal. In this case, T_i may be cancelled and its effects undone. Finally, a processor failure can halt the execution of T_i. When P is repaired a recovery algorithm rolls back T_i to its **INIT** state before it is restarted.

Successful execution means that T_i has received all necessary resources and its computations have completed. All actions have been recorded in the log and T_i is ready to commit. A successful execution moves T_i to an intermediate wait state (**W**). In the wait state T_i has waived its right to unilaterally abort. From the wait state, depending on the outcome of the commit protocol, T_i enters

the commit state (C) or is aborted. (Aborting a sub-transaction rolls it back to the initial state.) In the commit state locks on resources are released and the actions of T_i are made visible to other transactions.

Note that there is no transition out of the commit state. A transaction that is committed can never be aborted. This must remain true even when failures occur. Assuring that failures cannot cause state transitions is easily done by recording state changes in the log. Upon recovery, examination of the log determines the state of each sub-transaction at the time of the failure. The transitions of sub-transactions to either the abort or commit state must be coordinated by the commit protocol, so that either all sub-transactions commit or all abort. We first discuss the canonical commit protocol: the centralized two-phase commit. Then we discuss its variants.

2.2.1 Two-phase Commit

The canonical commit protocol is the two-phase commit with centralized coordinator. In this algorithm one node is designated coordinator and the other participating nodes are slaves. For simplicity we assume that the coordinator is located at the originating site of the transaction. The coordinator for transaction T does the following.

Commit Coordinator

- Phase 1. Broadcast an **EXEC** message to each sub-transaction of T. Write **WAIT** to the log and wait for a reply from each sub-transaction. (Each sub-transaction will send a reply only after it has completed its part of the transaction and written the appropriate **UNDO-REDO** log to stable storage.)
- Phase 2. If any slave sends a negative reply (**NOK**) or a timeout expires, then write **ABORT** in the log and broadcast **ABORT** to the slaves. This will cause the slaves to undo the transaction. Else if all the slaves respond positively (**OK**) then write a **COMMIT** entry to the log and broadcast a **COMMIT** message. This terminates the protocol for the coordinator.

A slave in the protocol does the following.

Slave:

• Phase 1. Wait for an **EXEC** message from the coordinator. If the sub-transaction completes successfully then write **WAIT** in the log and send an **OK** message to the commit coordinator. If unsuccessful, write **ABORT** to the log and send a **NOK** message to the coordinator. An abort terminates the protocol for the slave.

• Phase 2. Wait for the commit decision from the coordinator. If the verdict is **ABORT** then undo the transaction, write **ABORT** to the log and release all locks. If the verdict is **COMMIT** then write **COMMIT** to the log and release locks. This terminates the protocol for the slave.

Figure 7 shows the state diagrams for the coordinator and the slave protocol [Skeen82]. The label on each arc describes the transition. The top part of the label is the event(s) that triggers the transition; the bottom part is the message(s) that is sent out once the transition occurs. An asterisk means that a message must be received from all the slaves or sent to all the slaves.

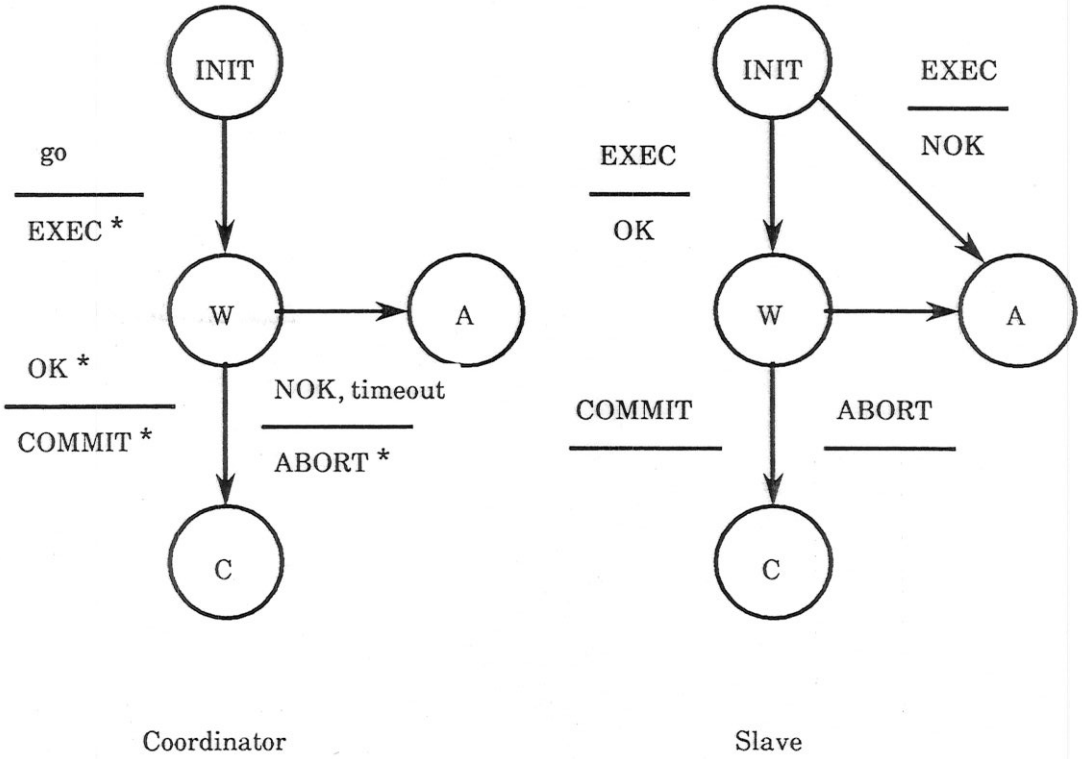

Figure 7. Centralized two-phase commit.

When a slave fails, it remains in its last state until the recovery. If the coordinator is in the **W** state, it can abort the transaction when it realizes that a node is not responding. If the coordinator is in the **C** state (i.e., the slave failed after it sent out the OK message), the transaction remains committed. When the slave recovers, it must establish communication with the coordinator to discover the outcome of the transaction. Fortunately, there can be no conflict between the state of the slave after the crash and the instructions received from the coordinator. If the coordinator says "Commit", then the slave must have responded OK before the crash and therefore is ready to commit its sub-transaction. If the coordinator says "Abort", then the slave could not have been in the **C** state, so it can still rollback the sub-transaction.

A failure of the coordinator, unlike that of the slave, may postpone the commit or abort decision until the failure is repaired. This can degrade system availability as the unfinished transaction **T** may interfere with the processing of other transactions because **T** holds resources that are needed by the other transactions. To illustrate, consider a transaction **T** which is executing at nodes **A**, **B** and **C**. Let node **A** be the coordinator for the commit protocol. Assume that **T** is in the second phase of the commit protocol; nodes **B** and **C** have sent their replies to **A**. Assume that node **C** voted 'OK'. The coordinator collects the replies and sends a commit (or abort) message to node **B** and then crashes. Node **B** receives the message from **A**, performs the indicated action and then crashes. Node **C**, the sole operational site, cannot safely terminate the commit protocol. If either node **A** or **B** aborted the transaction then **C** should abort, otherwise **C** should commit **T**. Because **C** does not know how nodes **A** and **B** voted, it cannot determine the appropriate action. The successful termination of **T** at node **C** depends on the repair of either **A** or **B**. Only then can **C** determine whether or not to commit **T**. A protocol which cannot be terminated successfully until failures are repaired is called *blocking*. Conversely, a protocol that can always be terminated, as long as the number of failures is finite, is termed *non-blocking* [Skeen82]. A non-blocking protocol will be studied in section 2.2.2.

The protocol we have described belongs to a group of two-phase commit protocols where one node is designated coordinator and the decision to commit is centralized. Other versions of two-phase commit exist within this group and are distinguished by their communication topologies. The class of

hierarchical commit protocols is obtained by generalizing the communication structure of the centralized version to a full tree. (The communication structure of the centralized commit protocol forms a tree of one level.) A single designated coordinator is at the root and messages flow along the links of the tree. The coordinator broadcasts to the slaves by sending a message to its immediate descendants. Each descendant similarly relays the message to nodes in its sub-tree. A leaf node sends its response to its parent. An internal node in the communication tree collects responses from its descendants and, based on these responses, sends a single message to its immediate ancestor.

A degenerate case of the hierarchical class is where each node has one parent and at most one child. In this case the communication topology is a simple linear chain. A phase in a linear commit protocol consists of a message propagating from one end of the chain to the other. The role of the coordinator alternates between the two nodes at the ends of the chain.

The centralized coordinator commit protocols are similar in that the number of messages needed to complete the protocol in the absence of failures is a linear function of the number of participating sites [Skeen82]. (The linear protocol uses the least number of messages.) The number of end to end message delays, however, varies greatly and is related to the depth of the communication tree. The centralized version has a shallow tree (one level) while the linear protocol forms a tree with as many levels as there are participating nodes. The number of end to end delays is a measure of the degree of parallelism of a protocol. In the centralized version nodes process and vote on a transaction in parallel. In the linear protocol sites process the transaction sequentially. Increased parallelism means that a transaction can be completed more quickly. Faster commit protocols are less likely to experience a failure before a transaction is committed.

The centralized coordinator may represent a potential bottleneck in the protocol. A heavily loaded coordinator might unnecessarily delay the termination of the protocol if it cannot process response messages as quickly as they arrive. Distributing the role of the coordinator is one way to eliminate the bottleneck. This leads us to the decentralized two-phase commit protocol:

• Phase 1. The originating site spawns the necessary subtransactions as before. When a cohort finishes its subtransaction it broadcasts its reply (OK, NOK) not just to the originating site but to all the other participating istes.

• Phase 2. Each site waits until it receives replies from all cohorts and then it appropriately commits or aborts.

Note that there is no central decision maker; each node independently commits or aborts the transaction based on the votes received from the cohorts. In the absence of failures all nodes will receive the same set of messages and make the same decision. However, if there are failures or timeouts, the participants cannot make unilateral decisions like the centralized coordinator did. Instead, each cohort must delay its decision until it gets all missing messages. If a message is missing, a cohort can request it from any node that has already received it.

2.2.2 Three-phase Commit

The three-phase commit protocol is the simplest non-blocking protocol. It is very similar to the two-phase commit, differing by an extra round of message passing. The states and transitions are shown in figure 8.

Coordinator:

Phase 1. Is the same.

Phase 2. If any slave replies to abort the transaction or failed before replying, the coordinator sends **ABORT** messages. Otherwise the coordinator broadcasts a **PREPARE** message directing the slaves to enter a committable state **P**. The slaves send an acknowledgement to the coordinator. (In two-phase commit the nodes moved directly to commit the transaction.)

Phase 3. When the coordinator has received acknowledgements from all the slaves, it sends a **COMMIT** message to the nodes. The slaves commit the transaction.

Slave:

Phase 1. is the same.

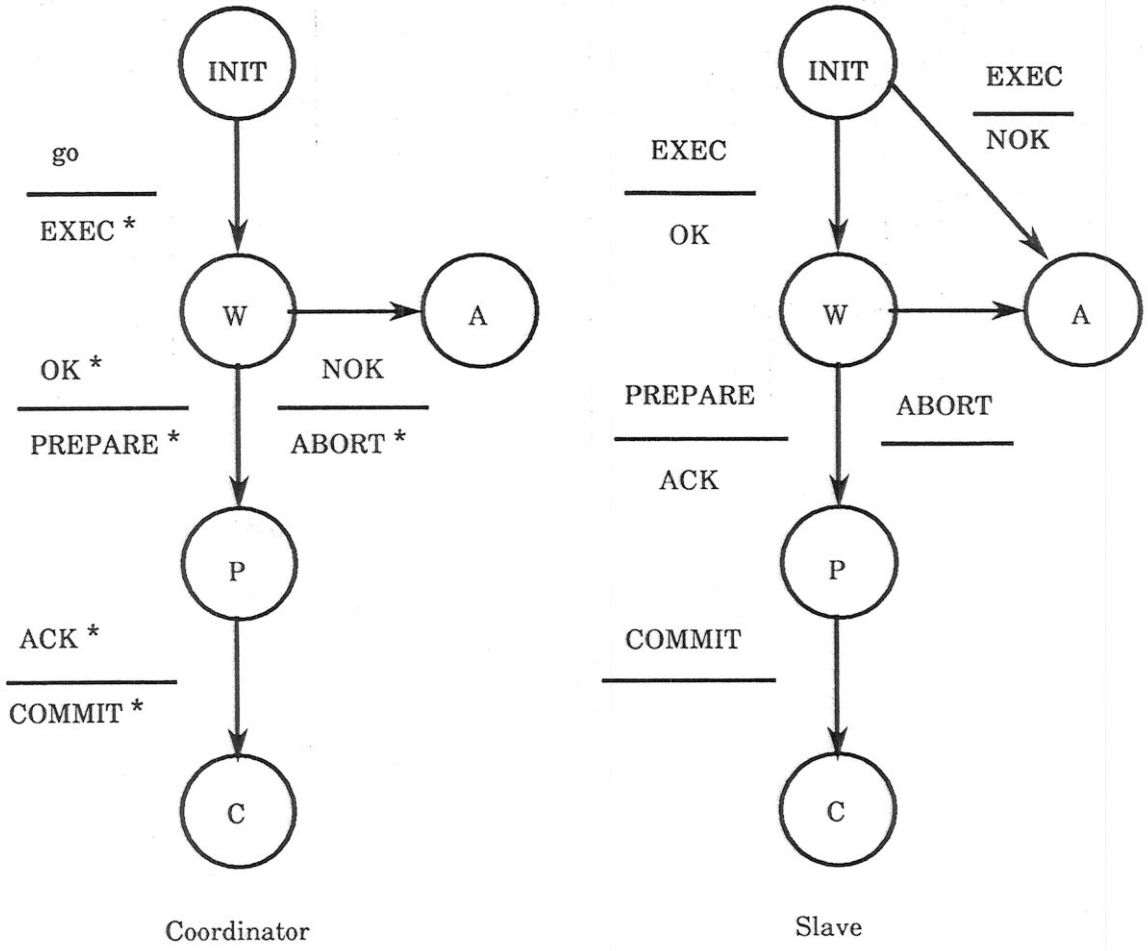

Figure 8. Three-phase commit

Phase 2. If the coordinator sends a **PREPARE** message then move to state **P** and send an acknowledgement (**ACK**) to the coordinator. Else if the coordinator sends an **ABORT** message, then abort the sub-transaction.

Phase 3. When the coordinator sends a **COMMIT** message enter the commit state (C) and commit the sub-transaction.

The algorithm can be altered to produce the hierarchical, linear and fully distributed versions of three-phase commit. Note that the extra phase of this protocol raises the cost of using it. Both the

total number of messages needed to commit and the number of end to end message delays are increased. The extra cost does buy us something, namely the non-blocking property. (Recall that a non-blocking protocol can always be terminated as long as the number of failures is finite.) We now discuss how a termination protocol takes advantage of the non-blocking property in order to safely complete a transaction after the coordinator node has failed.

2.3 Termination

A *termination protocol* is an algorithm that is invoked to recover from the failure of the coordinator node during the commit protocol. The purpose of the termination protocol is to safely terminate, by either committing or aborting, unfinished transactions at active slaves. (An active node is one that has never experienced a failure for the duration of the transaction.) The success of a termination protocol depends on whether the commit protocol is a non-blocking one. As shown previously, terminating a transaction under two-phase commit is not always possible. Under three-phase commit however, active slaves can always terminate a transaction. Here is why.

Consider the state of a transaction **T** as represented by the log information at the active nodes after some of the participating nodes, including the coordinator, have failed. Clearly, if any one of the active nodes has committed **T** then it is safe for the other active nodes to commit **T**. Similarly if any one of the active nodes has aborted **T** then it is safe for the other nodes to abort **T**. Likewise, if any one node is in the wait state **W** then it is safe for the nodes to abort **T**. This follows from the fact that all nodes must enter the prepared state **P** before any one node can commit the transaction. The fact that a node is in state **W** means that the transaction could not have been committed by any node and therefore can be safely aborted. Finally, if one of the nodes is in the prepared state **P** then **T** can be safely committed. This follows from the fact that a node enters state **P** only if none of the nodes voted to abort the transaction. The fact that a node is in state **P** means that all of the nodes had previously entered state **W**. Thus the transaction was not aborted by any of the nodes.

We can now design a termination protocol with the following guidelines. The protocol must use or continue the three-phase commit in order to protect against future failures. Here is an outline of a centralized version of the termination protocol.

Central site termination:

- Step 1. Elect a new coordinator for T from among the active nodes. Electing a unique coordinator in a distributed system is the subject of [Garc82a].
- Step 2. The new coordinator polls the active nodes to learn what state T is in at each of the nodes.
- Step 3. If any of the nodes committed T, send **COMMIT** messages to the active nodes. If any node has aborted, send **ABORT** messages. In either case the protocol ends.
- Step 4. Otherwise, if a **P** state is observed, send **PREPARE** messages to the active nodes that are not in state **P**, and continue the three-phase commit protocol as a coordinator in the wait (**W**) state. Similarly, if no **P** state is observed, then send **ABORT** messages to the active nodes.

If the new coordinator fails before T is completed then the termination protocol is invoked again. This can be repeated as long as there exist active nodes for T. On recovery a failed node must consult an active node to learn the outcome of T. Moreover, a failed node is not allowed to participate in the termination protocol if it recovers before the protocol has finished. The information that a failed node brings to the protocol is outdated and should not be used. This means that if *all* the participating nodes for T fail before the transaction has completed, then the system must wait until all the nodes recover before making a termination decision. Only then can the system determine the node or nodes which failed last and thus have the most recent state of the termination protocol. Also note that the worst case delay for successful termination involves **n-1** sequential coordinator failures where **n** is the number of participating nodes.

2.4 Partial Rollback

So far, the recovery strategy used to handle an aborted transaction T has been to completely **UNDO** all sub-transactions of T. This can be an expensive procedure especially if T has already consumed a lot of computer resources. A goal of recovery management is to minimize the amount of work that has to be undone and consequently redone.

Immediate savings on recovery costs can be realized from the recognition that complete rollback of a transaction might be unnecessary if one or few of the sub-transactions initiated the abort. Consider for example the transaction shown in figure 5. Let us say that sub-transaction T_4 was

unsuccessful and voted to abort. The obvious solution is to abort only T_4 and then restart it. The other cohorts remain in the wait state until the outcome of T_4 is known. If T_4, after several tries, is unable to complete successfully then the entire transaction is aborted and undone. This technique of aborting and restarting only some of the sub-transactions is known as *partial rollback*.

Partial rollback requires that the system maintain a record of the communication between sub-transactions. On restart the system retransmits the appropriate messages to the restarted sub-transaction. In the example the messages from T_3 would be resent to T_4.

Sometimes the rollback and restart of a failed sub-transaction can cause the rollback and restart of successful ones. This phenomenon is called *cascading rollbacks*. Again consider figure 5 and let us say that sub-transaction T_3 was unsuccessful. The restart of T_3 requires the restart of sub-transactions that either were spawned by T_3 or received messages from T_3. This includes T_4 and T_5.

A number of other techniques are closely related to partial rollback. For example, a *checkpoint* saves an intermediate state of a transaction so that processing can be resumed at that point. In our terminology, a sub-transaction (or a transaction) can be further decomposed into a sequence of smaller sub-transactions, where the end of a sub-transaction is a checkpoint. Recovery requires only that the transaction be backed up to the latest checkpoint. For example, an interactive transaction could establish checkpoints after each user message is processed. If bad input is discovered it would be sufficient to back up to the last checkpoint and re-request the last input. This is clearly preferable to undoing the entire transaction.

Similarly, in a *nested transaction* mechanism, transactions are decomposed into sub-transactions, but sub-transactions are restricted in the ways they can communicate [Moss81]. These limitations make it easier to decide what sub-transactions must be aborted.

We conclude this section by mentioning one more way to increase reliability without data replication. The method is primarily a hardware technique which uses dual-ported disks. A *dual-ported* disk is a storage device that can be controlled by either of two processors. The idea is that if the primary processor fails then the backup processor will start up and take control of the storage device. The data remains available and processing continues while the primary processor is repaired. For

this method to work, copies of the transaction programs and the transaction descriptors must be kept on stable storage and be available to the backup processor. The backup recovers by loading the appropriate programs and scanning the log to build the lock table and determine the states of transactions. Dual-ported disks are used in this fashion in several commercial systems, most notably in one built by Tandem [HC85].

Of course, reliability can also be improved by making copies of the data. This is the subject of the next section.

3. Data Replication

Failures in an unreplicated distributed database cause some data items to be unavailable to transactions. Throughput is decreased because transactions must wait until failures are repaired. Replicating data is the obvious way to increase availability and throughput. If a transaction is unable to access **X** at one node, it may be able to access it at another site.

Our model for a replicated database is as follows. Let **F** be some set of data items which are replicated in the same way. There exist anywhere from 1 to **n** copies of **F** where **n** is the number of nodes in the network. Each copy of **F** is stored at a different node. The set of all copies of **F** is called a *fragment*. Again the model of the distributed system is a collection of fail-stop computing nodes connected by a reliable, non-partitionable communication network. As the next section will show, more than just data replication is needed to tolerate the actions of fail-insane nodes.

Though data replication increases availability, it also creates some new problems for maintaining correctness. Chief among these is concurrency control. Access to a data item is no longer controlled exclusively by one node but rather is distributed across every node in the fragment. Concurrency control is now a distributed problem, the solution to which requires the cooperation of several sites. This cooperation leads to increased costs as the communication among participating nodes multiplies.

A second problem concerns ensuring that all nodes eventually receive all updates. Correctness requires that every copy of an data item contain the same value. Replication, however, allows transactions to continue to update an item even when some of the nodes in the fragment are down. Recovery management must provide a way for a recovering node to receive these "missed updates".

This section will discuss solutions to these and other problems. The goal is to achieve atomic execution of transactions and, at the same time, to take full advantage of the availability gained by data replication. We begin by examining transactions which operate on a single fragment. Later, we extend the discussion to include transactions which operate on multiple fragments.

3.1 Simple Strategies

One solution for concurrency control is to treat each copy of an item **X**, in a fragment **F**, as an independent data item. To read or write **X** a transaction must have locks on every copy of **X**. This is accomplished by spawning sub-transactions to execute at each node that stores a copy of **X**. The sub-transactions communicate with the local lock managers as described in section 2. Note that this solution simply multiplies the number of items that a transaction must read or write. As in the unreplicated case there is the strict requirement that all necessary items are accessible in order for a transaction to proceed. As before, distributed deadlocks may occur because transactions compete for shared resources. We assume that local lock managers cooperate to detect and resolve global deadlocks.

One benefit to this approach is that we can use the algorithms of the unreplicated case to solve the problems of distributed commit and crash recovery. Also, like the unreplicated case, a single node failure can render data unavailable. This is because a transaction must lock all copies in the fragment; one unavailable copy makes the entire fragment unavailable. Clearly this solution does not take advantage of data replication. Moreover costs are increased by the extra copies of the data and the additional sub-transactions needed to access the copies. Thus cost, not reliability has increased.

A somewhat better solution to the concurrency control problem is to have readers lock only a single copy of **X** and writers lock all copies. Simultaneous read and write operations will conflict at one node in the fragment. The lock manager at that node will serialize the conflicting transactions. Simultaneous writes will conflict at every node in the fragment. But only one transaction at a time can hold the locks for all copies of **X**. Thus write-write conflicts are also serialized. This approach also allows the use of mechanisms from the unreplicated case.

This method represents an improvement in data availability for read operations. Item **X** is available to a reader as long as one node in the fragment is active. Write operations, which must lock all copies, are still blocked by the failure of a single node in the fragment.

3.2 Primary Copy

There is another method to solve the concurrency control problem [Ston79]. Let one copy of a fragment **F** be designated the primary copy. The node that stores the primary copy is called the primary site. Let the other copies of **F** be called backup copies. Backup copies are stored at backup sites. A main feature of a primary copy solution is that transactions request write locks only at the primary site. Thus if there are **n** nodes in a fragment, the probability that a transaction blocks for a write lock because of node failure is exactly the probability that the primary has failed. This is likely to be far less than the probability that any node in the fragment has failed. This represents an improvement in availability over the previous strategy where the failure of any node would render the fragment unavailable.

We present two variations of the primary copy solution to concurrency control. They are distinguished primarily by how read locks are obtained and how the commit protocol is implemented. In the first, read locks are requested at the primary site and the data is also read at the primary site. In the second, transactions lock and read data at backup sites. Of course the failure of the primary site is a major problem because it renders the entire fragment unavailable. Initially, we assume that the primary copy is permanently fixed. Later in this section we consider methods to move the site of the primary copy from a failed node to one that is active.

3.2.1 Read Locks at Primary

In our first version of the primary copy scheme, lock management for a fragment is performed at the primary site. Transactions may execute at any node in the fragment but requests for read and write locks are sent to the primary site for approval. Moreover, the reading and writing of data values is performed at the primary site. Because lock management is centralized, all read-write and write-write conflicts will occur at the primary site and will be resolved by the lock manager to produce a serializable schedule.

When a transaction is ready to commit, it notifies the primary site which commits the updates to the database. At this moment the updates are reflected only in the primary copy of the fragment. The primary site, not the transaction, is responsible for ensuring that the updates are

propagated to the backup copies. It accomplishes this simply by sending messages to the backup sites with instructions to install the new values for the data items. A backup node installs the updates in an atomic fashion, in the same sequence they were generated.

The failure of a backup node is not difficult to handle as it is not the locus of lock activity. When a backup node recovers it must gather the updates to the database which it missed while it was failed. Sequence numbers are useful for accomplishing this. Since transactions are committed only at the primary site, we can assign a unique sequence number to each transaction that commits. When a backup node recovers it tells the primary node the sequence number of the last transaction recorded before the failure and requests the updates of all committed transactions with a higher sequence number.

There are a number of advantages to the primary copy strategy. One advantage is that it produces a single known sequence of updates to the database. Also the overhead of locking is reduced because transactions lock a single copy, not multiple copies. Another benefit to centralized locking is that distributed deadlocks are less likely. (Inter-fragment deadlocks between nodes can still occur, but deadlocks involving one fragment can all be detected at the primary node.) A disadvantage of centralized locking is that it places a large share of the workload at the primary site.

As we have described this strategy, read-only transactions must also request locks at the primary site. However, it is also possible for read-only transactions to execute entirely within the backup node. The idea is to perform all the writes of each update transaction as an atomic operation. This can be achieved with local locks (different from those at the primary site) in the same way a centralized database system ensures that (local) transactions are atomic. It is then possible for the read-only transaction to read local locks to obtain a consistent, though possibly out-of-date, view of the fragment. (For each update transaction, its changes to the fragment will either be fully reflected in the view, or they will be entirely invisible.)

3.2.2 Read Locks at all Nodes

Our second version of primary copy concurrency control is similar to the simple strategy discussed in section 3.1. Transactions are allowed to set read locks and read data at any node; write

locks are requested at "all copies". However, we now interpret "all copies" as all available copies, i.e., copies at functioning nodes. This of course improves availability: a writing transaction can complete even when nodes are down. However, it complicates data management because the set of nodes that have locks is changing dynamically.

To help cope with the changes we refer to a primary node. (A version of this strategy that does not use a primary site is discussed in section 3.3.) The main function of this primary site is to keep a list U that records the active nodes in the system. (We still assume that the primary is fixed.) We also use the primary site for backup recovery. That is, the primary will be in charge of giving a recovering node the ordered list of updates it missed.

When U is not changing, the protocol is straightforward. A transaction can get read locks and read data at any node in U. To update, the transaction requests write locks at and updates all nodes in U, including of course the primary. All read-write and write-write conflicts are hence avoided.

However, when nodes fail or come up, the primary changes U and complications can arise. For example, suppose that a transaction T requests read locks at a node P1, but then the node fails. When the primary removes P1 from U, other transactions will fail to request write locks at P1. Hence a conflict between T and the subsequent transactions may not be detected.

As another example of what can go wrong, consider the recovery of a node P2. It requests the missed transactions from the primary. The primary returns say $T_1, T_2, ..., T_n$ and adds P2 to U. At the same time a transaction T_{n+1} is in the process of committing. Since it has the old U set, it does not write at P2. Transaction T_{n+1} does write at the primary, but this occurs after $T_1, T_2, ..., T_n$ were sent to P2. Thus P2 installs the updates of $T_1, T_2, ..., T_n$ but misses T_{n+1}.

These and other problems can be avoided by carefully committing transactions. This works as follows. When a transaction T starts at node P1, it requests a copy of U from the primary site. Call the copy U'. While T executes, U may change and U' my become out-of-date. Thus, when T goes to lock at one of the U' sites, that site may be down. In this case, T can either abort or can request a new copy of the U set. (Note that when T starts, U' does not necessarily have to be obtained from the primary. For

instance T could use the U' list of the last transaction that executed at P1. In the infrequent case that U' is out-of-date, T will abort or get the latest U value from the primary.)

When T is ready to finish, it initiates its commit protocol. For simplicity we assume that this protocol is two-phase and coordinated by P1. (It could also be three-phase and/or distributed.) Node P1 sends phase one messages to all sites involved with T, i.e., all the sites in U'. (Unless T was a read-only transaction, it requested write locks at all U' sites.) When the primary site receives this message, it saves T's updates and acknowledges by sending U back to P1.

If P1 gets a U set that is different than U', then T is aborted. (As an alternative, T could be salvaged by requesting read and/or write locks from nodes in U - U'. We do not discuss this here.) If U' matches U and P1 gets positive acknowledgements from all sites involved, then T can be committed. This is because any node P not in U will see T when it recovers. If P recovers after the primary received and acknowledged T, then the primary will inform P of T's updates. If P recovered before the primary received T, then P must have been in the U set sent to P1, and hence is getting T's update directly from P1 through the commit protocol. So, since it is safe to commit T, P1 sends phase two commit messages to all participants.

When a node P recovers from a failure, it first disables all read and write lock requests. As discussed earlier, the missing updates are requested from the primary. The primary sends missed committed transactions $T_1, T_2, ..., T_n$, as well as the pending transactions $T_{n+1}, ..., T_m$ in the process of committing. Node P installs $T_1, T_2, ..., T_n$ in the database, and sets write locks for all the objects updated by $T_{n+1}, ..., T_m$. Since these transactions have not committed yet, P holds off their installation until the primary site later informs it of their outcome. At this point, P is open for business again and can start issuing read and write locks to new transactions.

When a node fails, obviously, it does not have to do anything. However, the primary must remove the node from the U list. (While the node is down and in the U list, transactions cannot complete.) The primary site can detect failures by periodically polling the backup sites or by requiring backups to send "I am alive" messages at fixed intervals. No response to a polling action or the omission of a message would indicate that a node has failed.

In this scheme, read-only transactions always see a consistent view of the data because updates are installed using local locks. Moreover, the view is up-to-date because updates are installed when a transaction commits.

We have presented two primary site mechanisms. The first centralized both concurrency control and crash recovery; the second only centralized crash recovery. The second mechanism distributes read locking to all nodes. This makes reading easier, but as we have seen, also makes writing harder and crash recovery more complicated. It is also worth pointing out that there are a number of variations and possible performance improvements of these two strategies.

3.2.3 Electing a new Primary

Up to now we have assumed that the primary site is fixed for all time. We now discuss what has to be done to select a new primary when the old one fails. When a primary node fails, all update activity ceases. Let us say that at this time the committed transactions are T_1, T_2, ..., T_n and the pending transactions are T_{n+1}, ..., T_m (i.e., these transactions have blocked during their commit). In recovering the system to a new configuration, two problems must be solved. First a new site must be chosen to be the primary site. And second, we must ensure that the new primary site as well as the active backups have seen the effects of all committed transactions, namely T_1, T_2, ..., T_n, and are aware of all pending transactions T_{n+1}, ..., T_m. The first problem can be solved by holding an election. Elections in a distributed system are discussed in [Garc82a]. The second problem requires that the effects of all committed transactions are permanent. The following example illustrates the problem of impermanent updates.

Let **F** be a fragment containing three copies of **X**, one copy being stored at each of nodes **P1**, **P2** and **P3**. Initially **P1** is the primary site, both **P2** and **P3** are failed and transaction **T1** updates **X** at **P1**. Then **P1** crashes and for some time all copies of **X** are unavailable. Later **P2** and **P3** recover, and **P2** becomes the new primary site. Since **P1** is still failed, **P2** and **P3** do not learn of the update by **T1**. Now **T2** updates the copies of **X** at **P2** and **P3**. When **P1** recovers, it updates its copy of **X** to agree with the value at **P2** and **P3**. The net result is that the update done by **T1** is not reflected in the state of the database. **T1** successfully committed but its actions were not permanent.

One solution to this problem is to require that the new primary site be a fully recovered node. By definition, a fully recovered node has seen all the update activity and therefore has the current state of the database. If there is no fully recovered node, then we must wait until all nodes in the fragment have been repaired. When this is done the nodes consult each other to determine which node was the last primary site and thus has the latest database state [Skeen83]. (In the example above, neither **P2** nor **P3** can recover since there is no primary site from which to gain the missed updates. Both must wait until **P1** recovers before processing **T2**.)

Another way to solve the problem of impermanent updates is to introduce a new commit protocol, the *majority commit*. To implement a majority commit we assign votes to each node in the fragment. As before, the commit coordinator will commit a transaction only if all participating sites agree to commit, with the additional proviso that the participating sites hold a majority of votes for the fragment. For example, if there are five copies of **X** and each node in the fragment has one vote then at least three nodes in the fragment must be active in order for a transaction to update **X**. Two nodes do not hold a majority of the votes and therefore are not allowed to update **X** by themselves.

As nodes fail and recover and new majority groups are formed, the majority commit assures that at least one node from the old majority group will be a member of the new majority. The fact that successive majority groups overlap means that updates are propagated from the old majority to the nodes in the new majority. Thus all updates are "remembered" and permanent. In our previous example, if **P1**, **P2** and **P3** have a vote each, **P1** will be unable to process **T1** if **P2** and **P3** are down. If instead **P1** and **P2** are up, then **T1** can execute. In this case, **P2** will have a record of **T1**. After **P1** fails, **P2** and **P3** can elect a new primary, since this primary, say **P3**, will get **T1** from **P2** before it starts processing new transactions.

An advantage of using the majority commit protocol over the previous method is that should all the nodes in the fragment fail we only have to wait for a majority of them to recover before processing can resume. The disadvantage is that a majority must exist in order to commit a transaction. The first method allows us to commit a transaction even when only one node is active.

On recovery the active backups plus the new primary must contain at least one fully recovered node or must have a majority of fragment votes before they can fully recover and resume processing. If the old primary failed during a commit protocol, then the new primary node initiates the termination protocol to finish pending transactions and create a new lock table (see section 2.3). When the old primary recovers, it does so as a backup copy. Since it is no longer the center of lock activity, the old lock table can be safely forgotten. Updates that occurred during the failure must be gotten from the new primary node.

3.3 Available Copies

Our first primary site algorithm centralized concurrency control and recovery. The second only centralized recovery management. If we distribute both concurrency and recovery control we arrive at the so-called Available Copies mechanism [Good83, BG83, BG85]. As before, read locks are requested at any available node, write locks at all available nodes. The difference is that the functions of the primary site are distributed.

In particular, the **U** set is now stored at all operational sites. When a node wishes to add itself to U, it runs a special transaction that updates all **U** sets as an atomic operation (i.e., it uses a commit protocol for this). This ensures that all copies of the **U** set are identical. When a node **P** fails, one of the remaining nodes runs a transaction to delete **P** from **U** atomically. Clearly, **P** is not involved in committing this transaction, even though it is a member of the original **U**. (If more than one node fails, a single transaction has to be run to update **U**.) Note that if the commit protocol to update **U** can block, then **U** and the entire system could be unavailable for an indefinite amount of time. When transactions commit, they utilize the same protocol described for the primary site version, except that now they can get **U** from any site.

When a node **P** recovers from a failure, there is no primary site to help, but this does not represent a problem. **P** can request the missed updates from any active node. If this node sends **P** all pending and committed updates up to the point that **P** is added to **U**, then **P** will not miss any updates. (Note that different nodes can observe different commit orders for two transactions **T1** and **T2**. This can only happen when **T1** and **T2** do not conflict with one another. Hence **P** can install them

in any relative order.) With this approach, any active node can help in recovery, so all active nodes must remember the updates they performed and their order.

If a recovering node finds no active node, then it must wait until all nodes are operational (or until a majority is operational, if a majority commit was used). As before, the group of recovering nodes must propagate to each other all committed (and pending) transactions before becoming active.

In summary, we have presented a number of strategies for managing replicated data within a fragment. (There are of course still other options and variations not discussed here.) Given the choices, selecting one approach for a given application or system is difficult. Each mechanism we have described has its advantages and disadvantages, i.e., there is no one that is best. The decision between centralizing and distributing the various functions is also hard. Distribution does not necessarily make the data more available (see section 3.1); centralization does not necessarily imply low availability (the primary site can move, as discussed in section 3.2.3). About the only general comment we can make is that as the protocols yield higher availability, they become more complex.

3.4 Multiple Fragments

So far we have discussed transactions that update a single fragment of data. To expand the techniques to handle multiple fragments the following two points must be considered.

1. Transactions must follow the concurrency control and crash recovery strategies for each fragment. Whether locking all copies or only the primary copy, the techniques for a single fragment must be applied individually to each fragment.

2. The commit protocols for each fragment must be integrated. When using the majority commit, a majority of each fragment updated by the transaction must agree to commit in order for the transaction to commit. This assures that updates are permanent in each fragment.

Two interesting problems arise when considering multiple fragments: the first concerns update transactions and the second read-only transactions. In the single fragment environment, an update transaction T that requested read locks at a node **P1** would also request write locks at **P1**. When there are several fragments, T can request read locks at **P1** and no write locks (it updates a fragment not stored at **P1**). We could now be tempted to exclude **P1** from T's commit protocol since it

has nothing to install in its database. The following example shows that this should not be done [BG83, BG85, Good83].

Consider a system with nodes **P1**, **P2**, **P3** and **P4**. Nodes **P1** and **P2** hold a copy of fragment **F1**; **P3** and **P4** hold **F2**. Using the available copies algorithm (section 3.3), a transaction **T1** read locks and reads **F1** at **P1**. At the same time, **T2** read locks and reads **F2** at node **P3**. Now both **P1** and **P3** fail. Next, **T1** decides to write **F2** and **T2** wishes to update **F1**. Transaction **T1** updates and commits **F2** at the only available site **P4**, and **T2** commits its changes to **F1** at **P2**. In a serial execution, this could have never occurred: either **T1** would have read the output of **T2** or vice versa.

This problem can be corrected by requiring a transaction like **T** to involve all sites that participated, including read sites, in the commit. In our example, **T1** would send phase one messages to **P1** (where it is holding read locks) and **P4** (where it is writing). When **T1** fails to receive an acknowledgment from **P1**, it realizes that its read locks there evaporated. Thus, there is no longer a guarantee that the data from **F1** is current. Hence, **T1** aborts. Transaction **T2** would run into a similar problem and abort also.

The second problem is that the execution of multiple-fragment read-only transactions may lead to non-serializable schedules under some of the concurrency control mechanisms we presented. However, the schedules will be weakly serializable [GW82]. An execution is *weakly serializable* if the schedule of the update transactions (ignoring the actions of the read-only transactions) is serializable and if every read-only transaction obtains a consistent view of the database (one that satisfies all consistency constraints.) We illustrate this with an example (see figure 9).

Consider a system with two nodes **P1** and **P2.** An item **A**, member of one fragment, is replicated at both nodes. Similarly, an item **B**, member of a different fragment, is located at both nodes. The fragment for **A** is managed with **P1** as a primary site (section 3.2.1). The **B** fragment is managed in the same way by **P2**. Let **T1** be a transaction that executes at node **P1** to update item **A**. Similarly, let **T2** be a transaction that executes at **P2** to update item **B**. **T1** and **T2** execute concurrently without conflict and are committed at roughly the same time. Because **T1** and **T2** execute independently and without conflict, there is no mechanism to force the schedule at **P1** to be

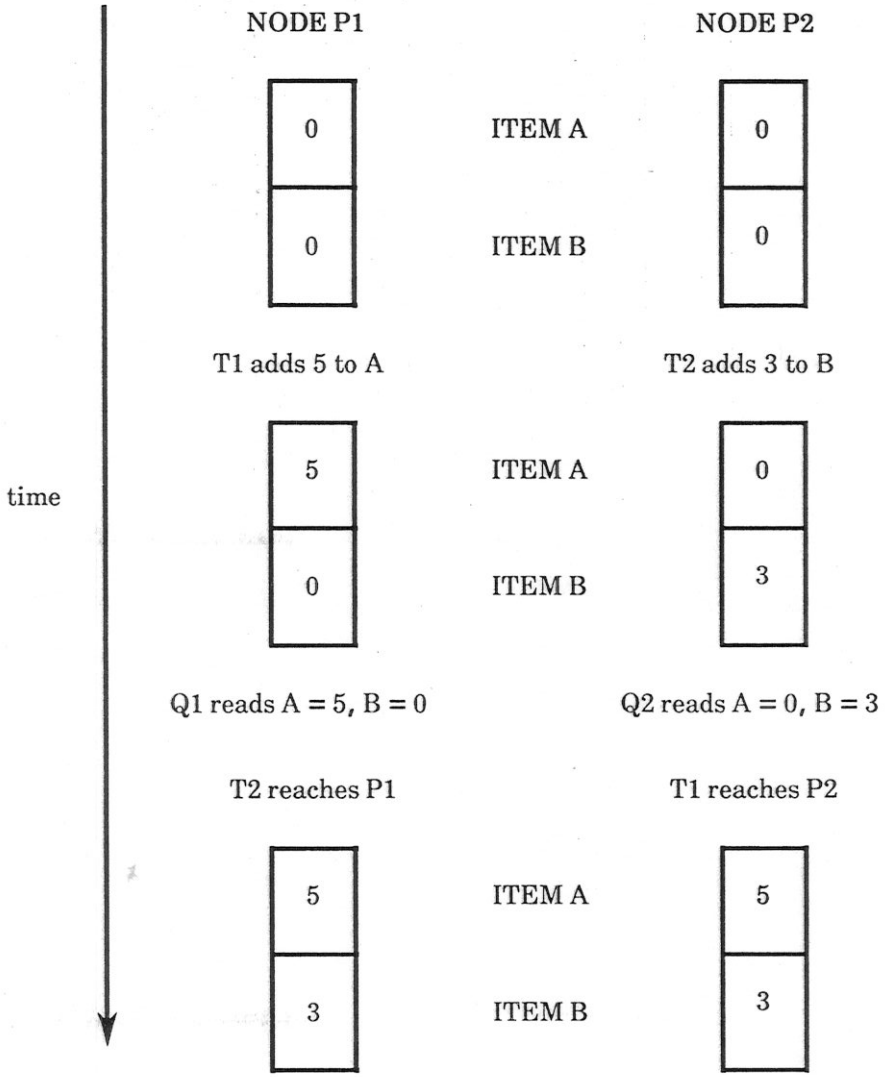

Figure 9. Weak serializability

the same as that at **P2**. Specifically, **T1** could commit at **P1** before **T2** and at node **P2**, **T2** could commit before **T1**. The central boxes show the database state after this occurs. At this time, identical queries **Q1** and **Q2** are submitted at sites **P1** and **P2** respectively. Since **Q1** and **Q2** are read-only transactions, they are allowed to read their data locally. **Q1** sees the effects of **T1** and not **T2** while **Q2** sees the effects of **T2** and not **T1**. Both **Q1** and **Q2** obtain consistent and correct views of the database. However these views could not have resulted from a single serializable schedule.

Weakly serializable schedules arise when read-only transactions are allowed to read consistent but possibly out-of-date data from a fragment. In the mechanisms discussed here, it occurs with a static primary that simply propagates updates to the backups (as in the example above), and with a dynamic primary that uses a majority commit (updates may be delayed to copies that are active but were not used to form a majority). Weak serializability also arises in other mechanisms [Chan85].

4. Full Replication

To this point we have considered reliability under a limited failure model, namely fail-stop nodes. These nodes fail in a clearly defined way, thereby simplifying the techniques needed to handle the failures. We now consider data processing in a distributed computing system composed of processing nodes that can fail in arbitrary ways, i.e., fail-insane nodes. Achieving reliability in a fail-insane environment requires techniques that are radically different from those presented in sections 2 and 3. We will first discuss the general solution, usually called n-modular redundancy. We then discuss how this solution can be applied to achieve a reliable database system. The communication network is assumed to be reliable and non-partitionable.

Clearly, tolerating fail-insane nodes is impossible in a system where the database is unreplicated or centralized. A single insane node could destroy its database; the damage would be irreparable. Neither can we allow one node to control all copies of the database, or even of a single item. What is needed to to tolerate insane nodes is complete data and transaction replication or *n-modular redundancy*.

The basic strategy of n-modular redundancy is shown in figure 10. **N** independent and identical tasks execute in parallel and send their outputs to one node called the voter. The voter selects the output that was produced by a majority of the tasks. N-modular redundancy requires $2m + 1$ copies or tasks in order to tolerate **m** insane failures. With **m** failures the voter can still detect the correct output from the majority $m + 1$ good nodes.

Task independence is an important aspect of n-modular redundancy. Each node has all the resources necessary to complete the assigned task. Both hardware and software are duplicated at every site. In the database world, this means that each node has a copy of the entire database. Resources are not shared because an insane node could gain control of a resource and refuse to relinquish it, thereby denying the resource to good nodes. Another important requirement is that all nodes receive the same input. Inputs and outputs and the associated problems are discussed in section 4.1.

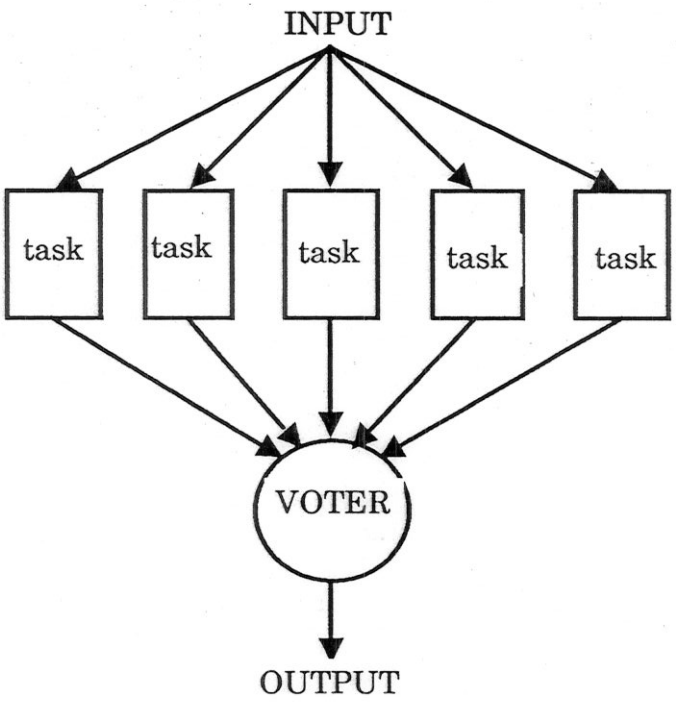

Figure 10. N-modular redundancy

N-modular redundancy has been used to design and build very reliable computing engines [Siew82]. More recently it has been used at the application level to develop reliable control systems. For example, the computers of the space shuttle use n-modular redundancy at both the hardware and the application levels [SG84]. This discussion concentrates on using n-modular redundancy to build reliable database systems. Note that n-modular redundancy is expensive, so we would expect it to be used on relatively small databases, e.g., a critical portion of a larger database or possibly a system directory.

As suggested, n-modular redundancy can be used at several levels. At the circuit level, figure 11, logic gates are duplicated to produce identical circuits which operate in parallel. The outputs are sent to a voter circuit. At a higher level, the method is applied to each of the main components (i.e., processor, memory, storage device) of a computer. A reliable component is built from the set of copies and a voter, figure 12. A reliable computer can be built using reliable components. Finally at the

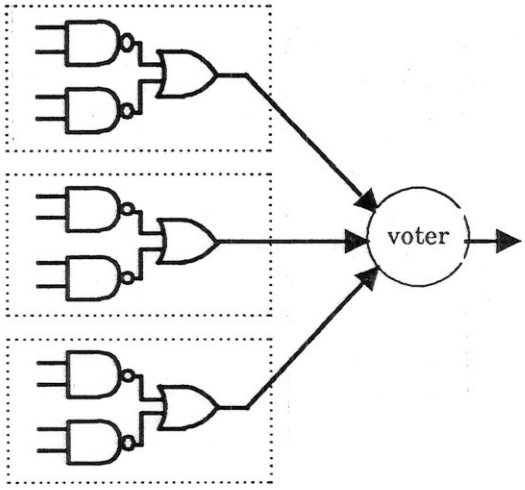

Figure 11. Circuit level redundancy

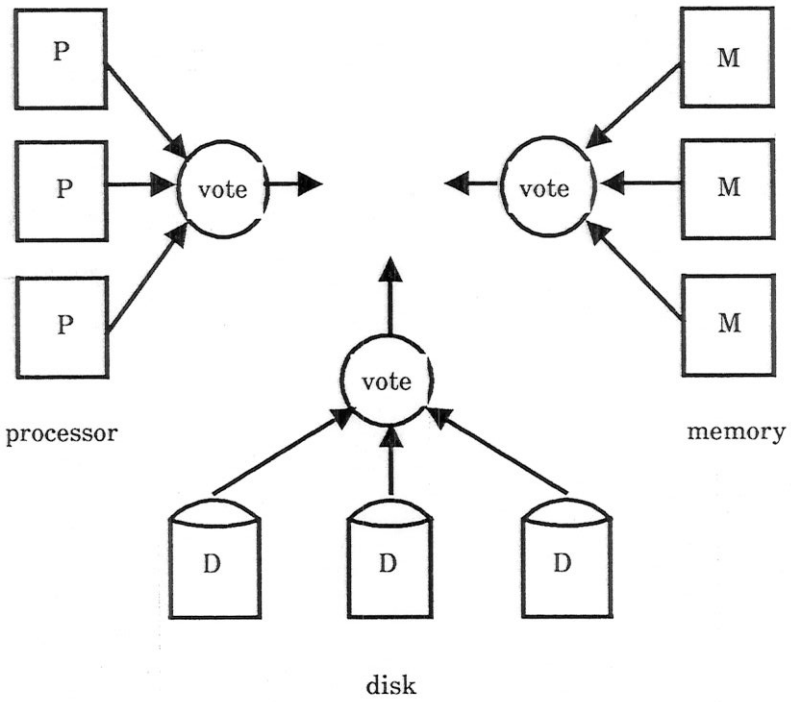

Figure 12. Component redundancy

application level, computers themselves are the unit of duplication. Computers operate

independently and in parallel on identical tasks. The outputs are sent to a voter which selects the majority, figure 13.

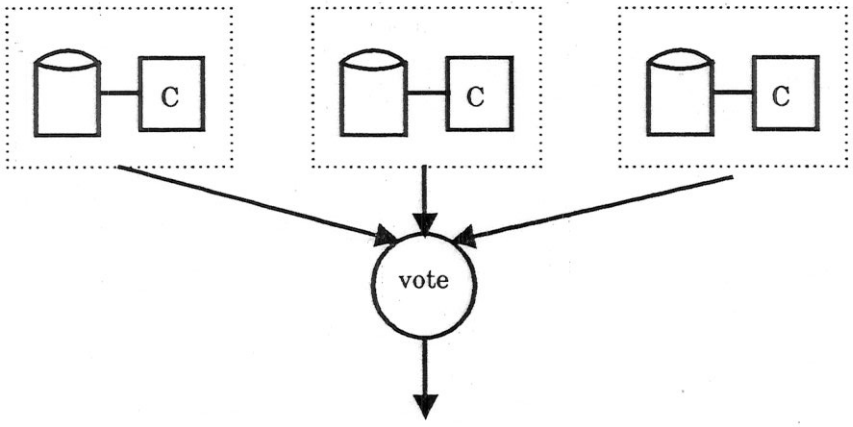

Figure 13. Application level redundancy

At which level to implement n-modular redundancy is a design choice influenced by a number of factors. Some of them are:

- Size of task output. If the tasks have small inputs and produce relatively small outputs, then a voter mechanism at the application (software) level could reduce communication overhead and would be efficient. If, however, tasks produce very large outputs then the voter mechanism would be more efficiently implemented at the computer or circuit level.
- Application independence. When a machine is built using n-modular redundancy at either the circuit or computer levels, the result is a reliable computer. No additional mechanisms are needed as the reliability is built into the hardware. Such a machine is general purpose; it guarantees the same level of reliability to all applications.
- Simplicity. Implementing n-modular redundancy at the circuit or computer levels requires special, hence more expensive, hardware. By contrast, n-modular redundancy at the application level can be achieved by using standard off-the-shelf computer components.

As stated earlier, we are interested in building reliable database systems. If we have n-modular redundancy at the computer level then all our problems are solved. We simply run a

centralized DBMS on the reliable computer. However, since in this paper we are interested in distributed data management, we will focus exclusively on application level replication. Furthermore, many database systems typically run transactions that have short descriptions (e.g., a relational language statement) and short outputs, and can benefit from this approach. Some of the distributed data management problems which warrant special attention are:

- Input and output (or voter) nodes. In database applications it is reasonable to consider the input/output nodes separately from the processing nodes. We will see that input/output nodes require a different failure model from processing nodes.
- Input agreement. In order for n-modular redundancy to work correctly, all processing nodes must execute the same transactions in the same order. In other words, execution schedules must be identical. This is necessary to ensure that a transaction sees the the same database state at each node and hence produces the same output.
- Recovery. Repairing an insane node enables the system to sustain additional failures at a later time.

We will now discuss these topics.

4.1 Input and Output Nodes

In many database applications input and output usually occurs at a *single* node that is separate and distinct from the processing nodes in the system. For example, in a banking system, most transactions are submitted by a teller at a personal terminal. The input/output node is now a critical component and cannot fail insanely. Otherwise it could invalidate the results of the processing nodes by conveying garbage to the users. An insane input node could submit bogus transactions.

This problem is easily "solved" by assuming that input/output nodes are always perfect. However this seems overly restrictive: so far we have required only that $\mathbf{m+1}$ out of $\mathbf{2m+1}$ processing nodes be perfect, yet now we are requiring that all input/output nodes be perfect.

There are basically two ways out of this dilemma. The first is to make the user be the input/output node. That is, the user could submit his transaction at $\mathbf{2m+1}$ different terminals. He

could also directly examine the **2m + 1** results of his transaction and perform the majority operation himself. This seems unsatisfactory for two reasons: firstly, the burden of failure should be placed on the system rather than the user. Secondly, users themselves are not always perfect.

The other alternative, and the one we take here, is to relax the failure model for the input/output nodes. Since an output node performs a very simple function it is easy to see what failures it could tolerate. It could lose or modify the outputs of some processing nodes as long as the majority, correct result is not altered. It could also crash giving the user his results later, or not giving them at all. The crash of an output node is acceptable. When a user fails to get the output, he can submit a query (directing the output to a different device) to see if his transaction committed and if so, what the results were.

The function of an input node is to encapsulate the user's transaction and distribute it to the processing nodes. We must require that an input node encapsulates a user transaction correctly and faithfully. It does not alter the transaction nor does it spontaneously generate bogus transactions. The input node can fail when distributing the transaction to the processing nodes. We will see that failures during distribution can delay the execution of a transaction but has no effect on database correctness.

4.2 Input Agreement

Input agreement is the next problem that we consider. Let us study a simple version initially. Suppose a single input node **I** at time 0 wishes to broadcast a single transaction **T** to **2m + 1** nodes for execution. Since we are using the fail-insane model, up to **m** of these nodes could be insane. In addition, the input node could also fail. We will only find two outcomes acceptable: either all nodes in a perfect state execute precisely **T**, or none of them do. This problem has been called the Byzantine Agreement (BA) problem. (Incidentally, the name "Byzantine" refers to a military scenario that was initially used to describe the problem [LSP83].) We will not study the BA algorithm formally or in detail but rather present the basic idea. We gradually arrive at the BA solution by first examining some methods that do not work. (After studying this problem, we return to the multiple input node, multiple transaction case.)

Clearly we cannot rely on input node **I** alone to distribute transaction **T** to the processing nodes. Node **I** could send **T** to the first processing node and then fail. Only one of the processing nodes has received and executed the transaction. Let's modify the method by adding one level of forwarding. Now **I** sends **T** to all the processing nodes, and each processing node forwards the transaction it receives to all the other processing nodes. As shown in figure 14 this method also fails to ensure that all good nodes receive the same value. **I** could fail after sending **T** to **P1** and **P1** could fail after sending **T** to only some of the remaining nodes. Thus some of the good nodes received and executed **T**, and some did not.

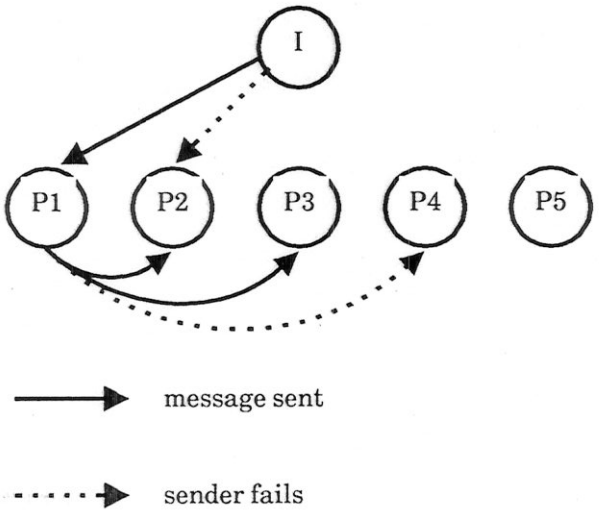

Figure 14. One level forwarding

To solve the problem, we can require that nodes like **P2** and **P3** forward **T** even when they get it from other processing nodes. This is what the BA algorithm does (or at least one version of it), but it also keeps track of the number of times a transaction has been forwarded in order to avoid "infinite forwarding". Before we describe how the algorithm limits the number of times a transaction is forwarded, we must introduce some new assumptions.

- Nodes in the perfect state have accurate and synchronized clocks. Specifically, at any instant the clocks differ at most τ time units. A signal from a very reliable clock can be periodically broadcast (by

radio or a dedicated line to avoid network failures) to synchronize the clocks. Alternatively, a reliable clock synchronization protocol can be used [LM84, LL84, HSSD84].

- The processing time of perfect nodes can be bounded. Given a sequence of code **s** we can compute the maximum amount of time t_s that a perfect node will take to execute the code.
- Messages are authenticated. All messages are signed and encoded by senders, in such a way that the receiver can determine unequivocally who sent the message and what it contained. A third node that simply forwards the message cannot alter it in any way. A node may refuse to forward the message. (Actually Byzantine Agreement is possible without authentication [LSP83]. The resulting algorithms are less efficient, though. Since authentication is practical and well understood, we will keep this assumption.)

From these assumptions and the guaranteed delivery time of reliable networks (section 1.4.2), we derive another:

- There exists a maximum send-receive-forward time Δ. If perfect node P_i sends a message to perfect P_j at time 0 then by time Δ, P_j has received, and signed the message and forwarded it to another node.

The basic solution works as follows: input node **I** signs a message containing a descriptor for transaction **T** and sends it to the processing nodes at time 0. When a processing node P_i receives a message it counts the number of signatures on it. If there are **s** signatures and the clock time is less than or equal to $s \cdot \Delta$ then P_i signs the message and forwards it to all nodes that have not signed it so far. (Since signatures cannot be forged, the presence of one indicates that the node already received the message.) If, however, the clock time is greater than $s \cdot \Delta$ then P_i ignores the message. Node P_i does this because the message arrived later than expected and thus must have come from an insane node. Messages from insane nodes are, of course, ignored. The rounds of message passing, one every Δ or less time units, continue until at most time $(m+2) \cdot \Delta$. (The usual BA algorithm stipulates $m+1$ rounds of forwarding. This would be the case if there were just processing nodes. In our situation the extra round of forwarding is actually the initial distribution of the transaction by the input node.) By time $(m+2) \cdot \Delta$, either all perfect nodes will have received **T**, or none will have. Thus at this time, if a node has received a transaction, it executes.

To see why this algorithm works, let us place ourselves in the situation of the fail-insane processing nodes, and let us try to subvert the algorithm. If any one of us sends **T** to a perfect node Δ seconds before the end of the algorithm (end is at time $(m+2)\cdot\Delta$), the perfect node will have sufficient time to forward **T** to all other perfect nodes. So then we must try to send **T** at the "last minute", i.e., Δ or less seconds before the end. However, we must send **T** in a valid message. At this stage, a valid message must have $m+2$ signatures, and they must all be from fellow failed nodes. (If a perfect node is in the list of signers, it would have already correctly distributed **T**.) Unfortunately, we are at most **m** failed processing nodes and one failed input node, so there are not enough to sign.

We have described a synchronous BA protocol; all the correct nodes begin the agreement protocol at exactly the same time. Recently, Bracha and Toueg [BT85] have described an asynchronous BA protocol. Because it is asynchronous it is possible that some correct nodes initiate the protocol and reach agreement while other nodes are not yet aware that the protocol has begun. Of course, all correct nodes will eventually agree on the same value.

There are many different solutions to Byzantine Agreement, but they all have the following two characteristics:

- In all algorithms, the worst case delay for reaching agreement is $(m+2)\cdot\Delta$ time units. However from a practical point of view, this delay is not critical because in most cases (when there are no failures) agreement can be reached much sooner. That is, the algorithm we sketched above can be modified so that a node processes a transaction as soon as it receives valid messages from all nodes in the system [DRS83]. Also note that the worst case delay for reaching agreement is similar to the worst case delay of the termination protocol for fail-stop nodes presented in section 2.3.
- In all solutions the message traffic is high, for the message received by a processing node from the input node must somehow be transmitted to all other processing nodes. This represents the essence of Byzantine Agreement: no single node is trustworthy, so all nodes must collect all information and make decisions for themselves. Batching several transactions in one message is one way to alleviate the high message traffic.

We make two observations about Byzantine Agreement:

1. Perfect nodes cannot agree on the identity of the insane nodes. A given node P_i may establish with certainty that some other node P_j is insane (because it failed to forward a message). However, P_j may appear perfect to other nodes, and P_i has no foolproof way of convincing these other nodes that P_j is indeed insane.

2. Any node can abort a transaction if the abortion is initiated by the transaction itself. This would occur, for example, when a transaction discovers there are insufficient funds to cover a withdrawal. The transaction would self-abort at all perfect nodes. However, a node itself may not abort a transaction because of local information e.g. resource deadlock. Consequently the transaction manager at each node must prevent such causes for abortion.

For the multiple input node, multiple transactions scenario, we can easily extend the solution we have presented. As we commented above, BA usually is initiated at fixed intervals. If an input node receives several transactions during an interval, it can package them together and broadcast them as a unit at the next BA. Having multiple input nodes does not present further complications either. All nodes can initiate BA at the same time and run the algorithm in parallel. At the end of each BA interval, the processing nodes will end up with **n** packages of transactions, where **n** is the number of input nodes. (If a node has no transaction to send, it sends a null package.) Next, the processing nodes sort the transaction in a pre-agreed upon way, so that all transactions are executed exactly in the same order at all perfect nodes. For example, the transactions from input node 1 could go first, followed by those of node 2, and so on.

The mechanism we have just described has been called the *state machine* approach [Lamp84, Schn82]. This is because the system behaves like a finite state machine: at each time interval, inputs are received and the database is transformed from one state to another.

4.3 Recovery from Insanity

Up to this point we have assumed that insanity is a permanent node property. However, it is desirable to repair insane nodes so that the system can then tolerate additional failures [GPD84].

Figure 15 illustrates on a three node system what we mean. In this case, only a single insane failure can be tolerated. Suppose that **P3** fails at time t_1. From that point on its database may be

ruined and its results are not trustworthy. If **P3** is not repaired, no failures of **P1** or **P2** can ever be tolerated. This is clearly not desirable.

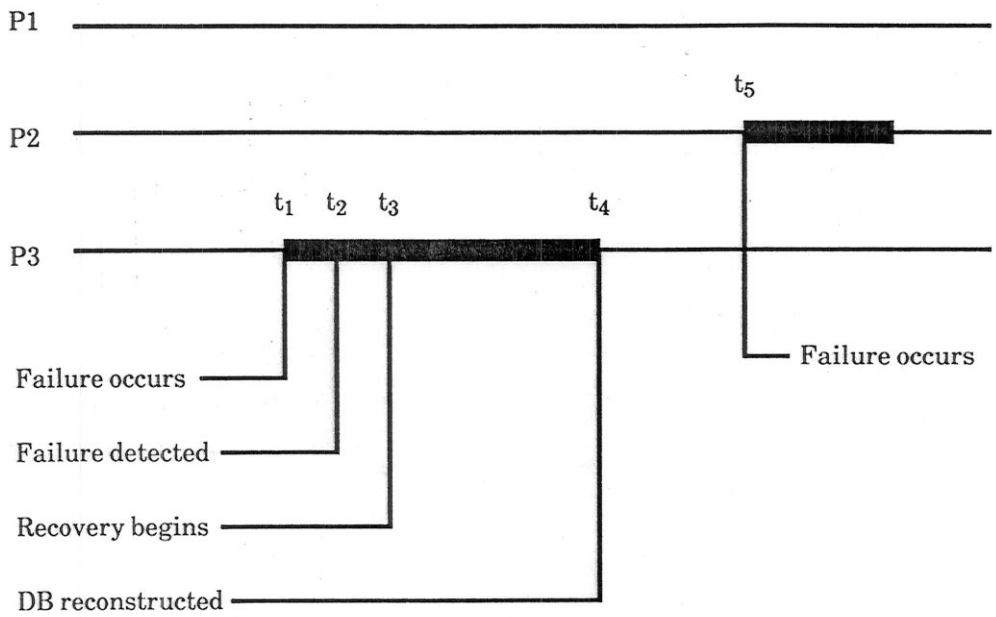

Figure 15. Recovery from insanity

As discussed earlier, the perfect nodes cannot be responsible for detecting an insane node like **P3**. Instead we must assume that **P3** detects its own failure. This is most easily accomplished by monitoring the outputs of all nodes. If its own output is different from that produced by a majority of the nodes, **P3** would identify itself as faulty. Alternately, the nodes could periodically compare their copies of the database. Of course, the comparisons could be made directly or through the use of database "checksums" or "signatures". In all cases node **P3** must detect its own failure.

By time t_2, the failure has been detected and **P3** must reconstruct its database copy. (**P3** may also have to repair other components, such as its clock or internal state tables, but we concentrate on the repair of the database itself.) Let us assume that BA intervals are numbered. First, **P3** selects the number of a future BA interval and broadcasts it to the other nodes. At the end of the appointed interval, each node takes a "snapshot" of its local database copy. This copy reflects all transactions

executed up to this BA interval. Since nodes process the same batch of transactions in each interval, the snapshots should be identical. The snapshots are sent to **P3**. Through voting, **P3** eliminates the effects of the other insane nodes, and then installs the new database locally. (In our 3 node example, voting is unnecessary since more than one insane node cannot exist.) Also, if **P3** can identify the portions of the database that are incorrect, only those portions need be copied [GPD84]. During the snapshot exchange period, t_3 to t_4, the perfect nodes must continue processing transactions, even if at a reduced rate. Insane failures should never halt the system.) Therefore, **P3** must record any transactions which arrive after time t_3, postponing their processing until time t_4. After time t_4, **P3** must catch up to the other nodes in the system. At any time after t_4, even before **P3** is fully caught up, the system can tolerate a second failure, say at time t_5.

5. Conclusions

In this paper we have studied the principles of reliable distributed database management. The algorithms we have presented make it possible to mask out or contain the effects of failed components, making a system reliable.

In closing, we would like to point out that there still remain many challenging problems in this field. In particular, we have not covered network partitions here. They make it difficult to achieve both correctness and availability. That is, to improve availability, we would like to allow local users to access copies of a database that are cut off from each other. However, doing this can compromise the correctness of the data [DGS84].

In addition, this paper has focused on coping with hardware failures. There are, of course, many other sources of problems, and a truly reliable system should also try to avoid or cope with these [RLT78, Gray85]. For example, many failures in commercial systems are caused by operator errors (the operator accidentally reformatted all disks), by bugs in the application code (the program to cancel a bank account forgot to check that the balance was non-zero), or by system bugs (the system got confused with its pointers). It is often harder to deal with these problems than the hardware failures that we have discussed here.

5. References

[BG81] Bernstein, Philip A. and Nathan Goodman, "*Concurrency Control in Distributed Database Systems*," Computing Surveys, vol. 13, pp 186-221, June 1981.

[BG83] Bernstein, Philip A. and Nathan Goodman, "*The Failure and Recovery Problem for Replicated Databases*," 2nd ACM Symposium on Principles of Distributed Computing, August 1983.

[BG85] Bernstein, Philip A. and Nathan Goodman, "*A Proof Technique for Concurrency Control and Recovery Algorithms for Replicated Databases*," Technical Report TR-85-21, Wang Institute, December 1985.

[BT85] Bracha, Gabriel and Sam Toueg, "*Asynchronous Consensus and Broadcast Protocols*," Journal of the ACM, vol 32, pp 824-840, October 1985.

[CP84] Ceri, Stefano and Giuseppe Pelagatti, *Distributed Databases, Principles and Systems*, McGraw-Hill, 1984.

[Chan85] Chan, Arvola, and Robert Gray, "*Implementing Distributed Read-only Transactions*," IEEE Transactions on Software Engineering, pp 205-212, February, 1985.

[DGS84] Davidson, Susan B., Hector Garcia-Molina, and Dale Skeen, "*Consistency in a Partitioned Network: A Survey*," Technical Report 320 Dept. of Electrical Engineering and Computer Science, Princeton University, August 1984.

[DRS83] Dolev, D., R. Reischug, and R. Strong, "*Early Stopping in Byzantine Agreement*," IBM Tech. Report RJ3915, June 1983.

[EGLT76] Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, "*The Notions of Consistency and Predicate Locks in a Database System*," Communications of the ACM, vol. 19, pp 624-633, November 1976.

[Garc82a] Garcia-Molina, Hector, "*Elections in a Distributed Computing System*," IEEE Transactions on Computers, vol. C-31, pp 48-59, January 1982.

[Garc82b] Garcia-Molina, Hector, "*Reliability Issues for Fully Replicated Distributed Databases*," IEEE Computer, pp 34-42, September 1982.

[Good83] Goodman, Nathan et al., "*A Recovery Algorithm for a Distributed Database System*," 2nd SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983.

[GPD84] Garcia-Molina, Hector, Frank Pittelli, and Susan Davidson, "*Applications of Byzantine Agreement in Database Systems*," Technical Report 316 Dept. Computer Science, Princeton University, June 1984.

[Gray79] Gray, J. N., "*Notes on Database Operating Systems*," *Operating Systems : An Advanced Course*, R. Bayer et al. editors, Springer-Verlag, pp 393-481, 1979.

[Gray85] Gray, J. N., "*Why Do Computers Stop and What Can Be Done About It?*," Tandem Technical Report, June 1985.

[GW82] Garcia-Molina, Hector, and Gio Wiederhold, "*Read-Only Transactions in a Distributed Database*," ACM Transactions on Database Systems, vol. 7 pp 209-234, 1982.

[HR83] Haerder, Theo and Andreas Reuter, "*Principles of Transaction Oriented Database Recovery*," Computing Surveys, vol. 15, pp 287-317, December 1983.

[HS80] Hammer, Michael and David Shipman, "*Reliability Mechanisms for SDD-1: A System for Distributed Databases*," ACM Transactions on Database Systems, vol. 5, pp 431-466, December 1980.

[HSSD84] Halpern, J., B. Simons, R. Strong, and D. Dolev, "*Fault-Tolerant Clock Synchronization*," Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, pp 89-102, August 1984.

[HC85] Horst, Robert and Tim Chou, "*The Hardware Architecture and Linear Expansion of Tandem NonStop Systems*," Tandem Technical Report 85.3, April 1985.

[IM80] Isloor, Sreekaanth S. and T. Anthony Marsland, "*The Deadlock Problem: An Overview*," IEEE Computer, pp 58-78, September 1980.

[Kent84] Kent, Jack and Hector Garcia-Molina, "*Performance Evaluation of Crash Recovery Mechanisms*," Tech. Report 329, Dept. of Computer Science, Princeton University, November 1984.

[Kim84] Kim, Won, "*Highly Available Systems for Database Applications*," Computing Surveys, vol. 16, pp 71-98, March 1984.

[Kohl81] Kohler, Walter H., "*A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems*," Computing Surveys, vol. 13, pp 149-183, June 1981.

[Lamp78] Lamport, Leslie, "*Time, Clocks, and the Ordering of Events in a Distributed System*," Communications of the ACM, vol. 21, pp 558-564, July 1978.

[Lamp84] Lamport, Leslie, "*Using Time Instead of Timeout for Fault-Tolerant Distributed Systems*," ACM Transactions on Program- ming Languages and Systems, vol. 6, pp 254-280, April 1984.

[LL84] Lundelius, J. and N. Lynch, "*A New-Fault Tolerant Algorithm for Clock Synchronization*," Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, pp 75-88, August 1984.

[LM84] Lamport, Leslie and P. M. Melliar-Smith, "*Byzantine Clock Synchronization*," Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, pp 68-74, August 1984.

[LSP83] Lamport, Leslie, R. Shostak, and M. Pease, "*The Byzantine Generals Problem*," Journal of the ACM, vol. 30, pp668-676, July 1983.

[LS79] Lampson, B. and H. Sturgis, "*Crash Recovery in a Distributed Data Storage System*," Xerox Research Memo, Xerox PARC, April 1979.

[MM79] Menasce, D. A. and R. R. Muntz, "*Locking and Deadlock Detection in Distributed Databases*," IEEE Transactions on Software Engineering, SE-5:3, 1979.

[Moss81] Moss, J. Eliot B., "*Nested Transactions: An Approach to Reliable Distributed Computing*," Ph.D. Thesis, MIT Dept of Electrical Eng. and Computer Science, April 1981.

[Ober81] Obermarck, R., "*Distributed Deadlock Detection Algorithm*," ACM Transactions on Distributed Computing, 7:2, 1982.

[RLT78] Randell, B., P. A. Lee, and P. C. Treleaven, "*Reliability Issues in Computing System Design*," Computing Surveys, vol. 10, pp 123-165, June 1978.

[RG77] Rothnie, J.B., and Nathan Goodman, "*A Survey of Research and Development in Distributed Database Management*," Third VLDB Conf., Tokyo, pp 48-62, 1977.

[Roth80] Rothnie, J.B., et al., "Introduction to a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems vol. 5, pp 1-17, March 1980.

[SS83] Schlicting, Richard D. and Fred B. Schneider, "*Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems*," ACM Transactions on Computing Systems, vol. 1, pp 222-238, August 1983.

[Schn82] Schneider, Fred B., "*Comparison of the Fail-Stop Processor and State Machine Approaches to Fault-Tolerance*," Dept. of Computer Science, Cornell University, November 1982.

[Siew82] Siewiorek, D. P., and R. S. Swarz, "*The Theory and Practice of Reliable System Design*," Digital Press, 1982.

[Skeen82] Skeen, Dale, "*Crash Recovery in a Distributed Database System*," Ph.D thesis, Electronics Research Laboratory, May 1982.

[Skeen83] Skeen, Dale, "*Determining the Last Process to Fail*," 2nd SIGACT-SIGMOD Symposium on Principles of Database Systems," March 1983.

[SG84] Spector, Alfred and David Gifford, "*Case Study: The Space Shuttle Primary Computer System*," Communications of the ACM, vol. 27. pp 872-900, September 1984.

[Ston79] Stonebraker, M. "*Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES*," IEEE Transactions on Software Engineering, pp 188-194, May 1979.

[TGGL82] Traiger, Irving, Jim Gray, Cesare A. Galteri, and Bruce Lindsay, "*Transactions and Consistency in Distributed Database Systems*," ACM Transactions on Database Systems, vol. 7, pp 323-342, September 1982.

[Ullm82] Ullman, Jeffrey D., *Principles of Database Sytems*, Computer Science Press, Inc., 1982.

[Verh78] Verhofstad, Joost S. M., "*Recovery Techniques for Database Systems*," Computing Surveys, vol. 10, pp 167-195, June 1978.