A Recovery Algorithm for a Distributed Database System*

Nathan Goodman
Dale Skeen
Arvola Chan
Umeshwar Dayal
Stephen Fox
Daniel Ries

Computer Corporation of America
Four Cambridge Center
Cambridge, MA 02142

Abstract

We describe a reliability algorithm being considered for DDM, a distributed database system under development at Computer Corporation of America. The algorithm is designed to tolerate clean site failures in which sites simply stop running. The algorithm allows the system to reconfigure itself to run correctly as sites fail and recover The algorithm solves the subproblems of atomic commit and replicated data handling in an integrated manner.

1. The Reliability Problem

Database systems use the concept of transaction to define correct behavior when many users share a database. A database system (dbs) makes two guarantees concerning transactions (1) If a transaction is unable to complete, all of its effects on the database are undone (2) Concurrently executing transactions will not interfere with each other. A distributed dbs (ddbs) may support replicated data, in which case a third guarantee is added (3) The copies of each logical data item will behave like a single copy for purposes of (1) and (2).

The reliability problem for a dbs is to implement transactions in the presence of failures. We identify two main subproblems. One, atomic commit, is the problem of attaining guarantee (1). The second subproblem involves the interaction of replicated data with guarantee (2) and is illustrated by the following example. Consider a database with logical data items X and Y and copies x_a, x_b, y_c, and y_d. T_1 is a transaction that reads X and writes Y, T_2 reads Y and writes X. Concurrency control is by two phase locking (2PL) [BG1,2, EGLT]. Replicated data is handled by the 'intuitive' algorithm to read a logical data item, a transaction may read any copy, to write, a transaction writes all copies that are up. The following execution obeys these rules, yet is incorrect, because the multiple copies do not behave like a single, logical data item.

$$r_1(x_a) \rightarrow \text{d-fails} \rightarrow w_1(y_c)$$

$$r_2(y_d) \rightarrow \text{a-fails} \rightarrow w_2(x_b)$$

('$r_1(x_a)$' denotes a read of x_a on behalf of T_1, 'd-fails' denotes the failure of the site storing y_d, etc. The arrows indicate the order in which events happen.)

Many reliability algorithms are known for centralized dbs's (cf [BGH, Gr, GMBLL, HR, Ve]), but only a few complete reliability algorithms are known for distributed dbs's (ddbs's). Many aspects of ddbs reliability have been studied, including atomic commit [ADEH, Ba, DS2, Ea, FLP, HS, La2, LS, ML, Re, Sk1,2, SkSt], site recovery [ABG, HS], resilient concurrency control for replicated data [ABDG, AD, Ea, Gi, MPM, Th, TGGL], site status monitoring [HS, Wa], Byzantine generals [Do1,2, DR, DS1,2,3, FFL, LSP, PSL], and network partition [FM, PR, St]. Resilience analysis of reliability algorithms includes [Co, CB]

This paper presents a reliability algorithm being considered for DDM, a ddbs under development by Computer Corporation of America [CDFLNR, CFLNR]. DDM is a general purpose ddbs that supports a high level, entity-relationship data model called DAPLEX [Sh]. Transactions are Ada programs with embedded high level data manipulation statements. The logical database can be fragmented, and each fragment stored at an arbitrary set of sites. Distribution and replication are invisible to the user At compile time, the system translates data manipulation statements (which, or course, reference the logical database) into statements that reference fragments. At run time, the system binds fragment references to specific fragment copies. A transaction can execute provided at least one copy of each referenced fragment is available.

The paper has seven sections Section 2 defines the types of failures our algorithm is designed to handle and the system architecture. Sections 3-6 describe the algorithm itself. Section 7 is the conclusion

* This research was jointly supported by the Defense Advanced Research Project Agency of the Department of Defense and the Naval Electronic System Command and was monitored by the Naval Electronic System Command under Contract No. N00039-82-C-0226.

2. System Model

2.1 Failure Assumptions

The sites of a distributed system can fail in many ways. The simplest site failures are *clean failures* in which a site simply stops running. The hardest failures are *traitorous failures* in which a site continues to run, but performs incorrect actions. Most real failures lie between those extremes. After a fault occurs, the site runs incorrectly until the fault is detected, whereupon the site stops.

We assume that all faults are detected before serious damage is done so that from the system standpoint all site failures are clean Also, when a failed site recovers, it 'knows' that it failed and can initiate a recovery procedure. (These assumptions are implicit in all centralized dbs reliability algorithms [GMBLL].)

While a site is down, other sites must be able to detect this fact [FLP]. In early days of the Arpanet, the network implemented failure detection internally, but today's networks do not offer this service. As a practical matter, the only mechanism available for detecting site failures is *timeouts*. For purposes of this paper, we assume that some failure detection mechanism exists, but do not specify which one. We assume that the mechanism is foolproof if the mechanism declares a site down, then indeed the site has failed. (This assumption is reasonable in the case of the early Arpanet mechanism. It is less reasonable for timeouts.)

Most network errors, e.g., lost, duplicate, or garbled messages, are handled by standard network software and are not considered here. From our standpoint, the only network failures are *partitions* in which two or more running sites are unable to communicate. Our algorithm is *not* designed to handle partitions.

To summarize The reliability algorithm described in this paper is designed to handle an arbitrary number of clean site failures It assumes that site failures are detectable by other sites. It is not designed to handle network partitions.

2.2 System Architecture

The system consists of four levels of virtual machine.

DDBS Functions
Process Incarnations
Processes
Sites

The bottom level contains *sites* (i.e., computers) connected by a computer-to-computer network. Next are *processes* connected by a process-to-process network. A process is a concurrent program running at a single site. On top of this we implement *process incarnations* connected by a corresponding network. A process incarnation is one 'lifetime' of a process starting when the process recovers from failure and ending when it fails again. The top level supports standard ddbs functions data managers (DMs) and transactions managers (TMs).

The bottom two levels are standard and we do not describe their implementation. Section 2.3 defines the behavior of the process level. Sections 3-6 describe the remaining levels

We model a computation as a partial order of *events*, using Lamport's *happens-before* partial order [La1]

The logical database is a set of *logical files*, each of which may be stored at any number (≥ 1) of sites. The copies of a logical file are called *physical files*.

A transaction is a program that starts with Begin, ends with End, and contains Read and Update commands referencing logical files.

2.3 Processes

A process exists in two states, *up* and *down*. An *up* process is one that is running correctly. When a process fails, it enters the *down* state where it does nothing. (A process is also *down* before it is initiated.) Later, the process can recover and return to the *up* state. When a process recovers, it 'knows' that it failed and executes a specified recovery procedure. Each process has some *stable storage* whose contents are unaffected by failures.

We posit the existence of a failure detection mechanism that lets an *up* process determine the state of another process.

Processes interact by sending messages through the network. If process p sends message M to process q there are three possible outcomes (1) q receives M. (2) q fails and p is notified that q has failed. (3) p fails. Note that if q does not receive M, p is aware that a failure occurred (either p failed, q failed, or both).

3. Process Incarnation

The process incarnation level synchronizes process failures and recoveries. This level lets higher levels act as if failures and recoveries happen sequentially in a well-defined order.

We define the behavior of this level in Section 3.1 and describe its implementation in Section 3.2. Section 3.3 treats the special case of total failure.

3.1 Functionality

A process incarnation (or simply, incarnation) exists in four states with the following transitions.

dormant —> recovering —> in —> out

A *dormant* incarnation does nothing. A *recovering* incarnation can interact with other parts of the system (e.g., to bring its database up to date) but cannot process user transactions. An *in* incarnation is fully operational. An *out* incarnation is 'dead' and does nothing. Once an incarnation is *out* it never again participates in the system. An incarnation goes *out* when its process fails or is brought down for reasons such as maintenance. Incarnations of a given process are totally ordered, each incarnation remains *dormant* until the preceding ones are *out*. Incarnations of a given process share the same stable storage to pass information from one incarnation to the next.

Transitions to _in_ or _out_ are governed by _status transactions_, Include and Exclude. A status transaction may be invoked by any _in_ incarnation and informs all _in_ incarnations of the state change. Include(i) informs all _in_ incarnations that i is _in_ and tells i all status information known to the invoker. Exclude(i) tells all _in_ incarnations that i is _out_. An incarnation is _in_ (resp. _out_) once any incarnation knows the state change.

The system executes status transactions serializably. More precisely, the system forces a total order over status transactions, say $s_1, s_2, \ldots, s_n$. Each incarnation i executes a (possibly empty) subsequence $s_i, s_{i+1}, \ldots, s_j$ where s_i is Include(i), s_j is before or is Exclude(i), and all database operations executed by i come between s_i and s_j in the happens-before partial order. Section 3.2 explains how we achieve this property.

The transition from _dormant_ to _recovering_ does not need a status transaction. A _dormant_ incarnation can enter the _recovering_ state any time after its previous incarnation is _out_.

Incarnations of different processes interact by sending messages through the network. If incarnation i sends message M to j, there are four possible outcomes (1) j receives M. (2) j is Excluded. (3) i is Excluded. (4) Total failure -- the processes of all _in_ incarnations fail.

Let us consider the above functionality from the standpoint of an individual process, p. When p recovers, its next incarnation, i, begins to execute in the _recovering_ state. Incarnation i may remain in this state for some time. Eventually, when i decides to be Included, it finds an _in_ incarnation, i', and requests that i' invoke Include(i). (If no _in_ incarnation exists, this is a total failure. See Section 3.3.) Incarnation i' invokes Include(i), thereby moving i to the _in_ state. When p fails, some _in_ incarnation, i', invokes Exclude(i).

Each incarnation maintains a _status database_ telling the status of all incarnations known to it. The database is updated by status transactions and by the receipt of messages from _recovering_ incarnations.

The incarnation level provides the following functions for higher levels of the system.

- _Retrieve_ from status database.
- _Watch_ for a specified state change. The higher level is interrupted when the status database is updated in the specified manner.
- _Broadcast_ message M to a set of recipients. The recipients are incarnations and may be in any state. Each _in_ recipient is expected to generate a response. The broadcast completes when all _in_ recipients have acknowledged M, and all other recipients have either been Excluded or not yet Included.

Broadcasts are synchronized with status transactions to achieve the following property. Consider a broadcast, b, invoked by incarnation i, and let $s_i, s_{i+1}, \ldots, s_j$ be the sequence of status transactions executed by i. The broadcast can be inserted into the sequence, e g., as $s_i, s_{i+1}, \ldots, s_{i+k}, b, s_{i+k+1}, \ldots, s_j$ such that i gets acknowledgements from all recipients whose Include precedes b and whose Exclude follows b (or does not appear) Section 3.2 explains how we achieve this property.

3.2 Implementation

We now describe the implementation of the incarnation level in terms of the process level.

Each process has an _incarnation number_ stored on stable storage which is incremented each time the process recovers. The combination of a process name and an incarnation number uniquely identifies an incarnation.

Each message sent between incarnations carries the incarnation numbers of the sender and intended recipient. Call these the send-number and receive-number, respectively. If a process receives a message with an old send-number, this indicates that the message has been adrift in the network for a long time, and is no longer applicable. In this case, the recipient ignores the message. If a message has an old receive-number, this indicates that the recipient process failed and recovered without the sender noticing the failure. In this case the recipient sends a response indicating the failure.

A process, p, may discover that another process has failed either directly via the system's failure detection mechanism or indirectly by the mechanism of the previous paragraph. When p discovers the failure it invokes an Exclude transaction, unless the Exclude is already underway.

Status transactions execute using a variant of Skeen's atomic broadcast protocol [Sk1,2, SkSt]. We describe the Include transaction, Exclude is similar

Include(i) invoked by incarnation j

Step 1.

- Incarnation j broadcasts 'Prepare-to-Include(i)' to all _in_ incarnations including itself.
- Each recipient treats the message as a request for an _Include lock_ on i. The recipient grants the lock and acknowledges the message unless it is already holding an Include or Exclude lock on any incarnation. Include locks also conflict with Broadcast locks, defined shortly. Deadlocks are, of course, possible here. A non-preemptive deadlock avoidance scheme, like Wait-Die [RSL], is a suitable way of handling these deadlocks.
- This step completes when all recipients have acknowledged the message or failed.

Step 2.

- Incarnation j sends its status database to i, and broadcasts 'Include(i)' to all _in_ incarnations including itself and i.
- Each recipient updates its status database, and releases the lock set in Step 1.

If j fails before completing the transaction, a variant of Skeen's distributed termination protocol [Sk1,2, SkSt] is run. Define incarnation k to be *incomplete* relative to the transaction if k is holding the Include lock set by the transaction, and k's process has not failed. An incomplete incarnation simply reinvokes the transaction from the beginning. The messages sent in Step 1 indicate that this is a reinvocation. A recipient holding a lock from an earlier invocation lets the new lock preempt the earlier one. A recipient that completed the earlier transaction acknowledges the message immediately without setting a lock.

The status transaction algorithm and termination protocol achieve the following properties. (1) If any incarnation completes the transaction, then every *in* incarnation completes the transaction or fails before the transaction completes. (2) Status transactions are totally ordered.

Broadcasts execute with a weaker protocol that synchronizes them relative to status transactions but does not attempt termination if the invoker fails

Broadcast M to set I invoked by incarnation j

Step 1

- Incarnation j locally sets a Broadcast lock on I. This lock conflicts with an Include lock on any member of the set I.
- Incarnation j broadcasts M to all *in* members of I.
- Each recipient acknowledges M. Recipients do not set locks.
- This step completes when all recipients have acknowledged the message or have been Excluded.

Step 2

- Incarnation j releases its Broadcast lock.

3.3 Recovery from Total Failure

A total failure has occurred when all *in* incarnations have failed

Normally, when a process recovers from failure its next incarnation begins to execute and finds an *in* incarnation to Include it. If the incarnation cannot find an *in* incarnation, it assumes a total failure has occurred, and the new incarnation stops running. The previous incarnation resumes, and runs the LAST SURVIVORS algorithms described below. (See also [Sk3].)

The LAST SURVIVORS algorithm calculates the set of incarnations that failed last. An incarnation is in this set if it has been Included, but not Excluded. The algorithm, run by incarnation i, maintains four sets.

S= {incarnations j that i has heard from while running the algorithm}

ALL= {incarnations k | some j in S has Included k}

OUT= {incarnations k | some j in S has Excluded k}

IN = ALL - OUT

= {incarnations k | some j in S has Included k and no j' in S has Excluded k}

The algorithm initializes these variables to

S = {i}

ALL = {k | k's state is *in* or *out* in i's status database}

OUT = {k | k's state is *out* in i's status database}

IN = ALL - OUT

Recovering processes exchange messages indicating the current values of S, ALL, and OUT. When i receives such a message, containing say S', ALL', and OUT', it updates its variables.

S = S U S'

ALL = ALL U ALL'

OUT = OUT U OUT'

IN = ALL - OUT

It can be proved that when $S \supseteq IN$, then IN is the desired set of last survivors, call this set LAST. Also, all incarnations that run the algorithm calculate the same value of LAST.

When LAST is calculated, i updates its status database to show all members of LAST to be *in*, and all other non-dormant incarnations *out*. If i is in LAST, it resumes normal operation. Otherwise, i was not a last survivor and stops running. To resume operation, i's next incarnation must be Included in the normal way.

4 Data Managers

Data managers (DMs) store and manage the database. This section describes DM operation under normal conditions Section 6 considers DM failures and recoveries.

Each DM stores a single physical file. When no confusion is possible we blur the distinction between a DM and the file it stores. For each logical file X, the set of DMs that store the copies of X forms a *logical DM* for X.

The state of a DM (or, equivalently, the file it stores) is the state of its incarnation. An *in* file has two substates, *online* and *offline*. On *online* file is up-to-date and can be used for transaction processing An *offline* file is not up-to-date, *offline* is a transient state through which a DM passes during recovery. A *recovering* DM is always *offline*. For other DM states the substate is irrelevant.

An *online* DM x processes the following operations. $Read_t$.

- Retrieve a portion of x on behalf of transaction t.
- $Update_t$. Modify x on behalf of transaction t. The update is not permanent at this time and may be undone by a subsequent $Abort_t$. The operation also creates an *update log* (similar to a REDO log [Gr]) containing enough information to perform the update on other copies of the file. If

transaction t updates x more than once, the update logs are collected into a single log. The update log may be distributed to the other copies of the file in the background while transaction t executes, or when t ends.

- End_t. If transaction t has updated logical file X, the DM obtains the update log and applies it to the database. The update is not yet permanent. Otherwise, the operation has no effect.
- $Commit_t$. Install t's updates permanently in the database.
- $Abort_t$. Undo t's updates.

The DM performs these operations under the command of the transaction manager (TM) controlling transaction t. When the DM completes an operation it returns a positive response to the TM. Occasionally, the DM may reject a Read, Update, or End operations, e.g., because of deadlock. When this happens the DM returns a negative response to the TM who then aborts the transaction (see Section 5). Commits and Aborts can never be rejected. Once a DM performs an operation on behalf of transaction t, the DM Watches t's TM. Section 5 explains what happens if the TM fails. The Watch is turned off when t commits or aborts.

Concurrency control is by basic distributed two phase locking (2PL method 12 of [BG1]). Locks are held until Commit or Abort.

Each DM maintains a recovery log containing enough information to bring an offline copy up-to-date. The recovery log contains (1) A committed transaction list (CTL) consisting of transaction identifiers for all transactions that have committed at the DM. (2) An aborted transaction list (ATL) analogous to the CTL. (3) A pending transaction list (PTL) identifying transactions that have executed at the DM but are not yet committed or aborted. (4) The update logs for all transactions whose End operation has been executed at the DM.

An offline DM x processes a single operation, Rollforward. The DM obtains the recovery log stored by some online member of its logical DM. DM x applies the update log to its database and updates its CTL, ATL, and PTL accordingly.

5. Transaction Managers

Transactions managers (TMs) control transaction executions. This section describes TM operations under normal conditions. Section 6 considers TM failures and recoveries.

Each transaction, t, issues all of its operations to a single TM. The TM binds the logical files referenced by t to physical copies that are available when t executes. The TM also coordinates atomic commit and abort. TMs are grouped into logical TMs analogous to logical DMs. The members of a logical TM serve as backups for each other during atomic commit, and store replicated copies of committed, aborted, and pending transaction lists (CTLs, ATLs, and PTLs).

TMs exist in online and offline substates, defined as for DMs (see Section 5).

An online TM supports the following activities.

File binding. For each logical file, X, referenced by transaction t, the TM selects a physical copy, x. The set of physical files selected for t is called its materialization.

Materialization watching. The TM watches each file in t's materialization using the Watch function of the incarnation level (see Section 3). If any file is Excluded before Phase 2 of atomic commit (define below), the TM aborts t.

Abort. The TM executes an Abort transaction analogous to the status transactions of the incarnation level (see Section 3).

Step 1.

- The TM broadcasts 'Prepare-to-Abort(t)' to the members of its logical TM using the incarnation level Broadcast.
- Each recipient tries to set an Abort lock on t. Abort lock on t conflict with each other and with Commit locks on t (defined shortly). A Wait-Die scheme [RSL] can be used to prevent deadlocks.
- This step ends when the Broadcast completes, that is, all recipients have acknowledged the lock or been Excluded.

Step 2.

- The TM sends 'Abort(t)' to all in members of its logical TM and all in DMs who were sent any operations for t.
- Each TM recipient updates its ATL and PTL, and releases the lock set in Step 1.

If the TM fails before completing the Abort transaction, a termination protocol like that of Section 3 is invoked.

The TM can abort a transaction at any time until Phase 2 of atomic commit begins. Thereafter, the commit algorithm governs all aborts.

Atomic commit. We use a variant of three phase commit [Sk1, Sk2, SkSt].

Phase 1.

- For each logical file X that t updated, the TM broadcasts End_t to all copies of X using the incarnation level Broadcast function.
- Each DM processes End_t as described in Section 4 and responds positively or negatively to the TM.
- This phase ends when all recipients have responded or been Excluded. There are two possible outcomes. If any DM responded negatively, or if any DM in t's materialization has been Excluded, the TM aborts t. Otherwise it continues the commit protocol.

Phases 2 and 3 constitute a Commit transaction virtually identical to the Abort transaction.

Phase 2.

- The TM broadcasts 'Prepare-to-Commit(t)' to the members of its logical TM using the incarnation level Broadcast.

- Each recipient tries to set a Commit lock on t. Commit locks on t conflict with each other and with Abort locks on t.
- This phase ends when the Broadcast completes, that is, all recipients have acknowledged the lock or been excluded.

Phase 3.

- The TM sends 'Commit(t)' to all in members of its logical TM and all in DMs who were sent End's in Phase 1.
- Each TM recipient updates its CTL and PTL, and releases the lock set in the previous step.

If the TM fails (and is excluded) before committing or aborting transaction t, there are three cases. (1) Some in member of the logical TM has received the 'Prepare' message but not the 'Commit' or 'Abort'. (2) No in member has received the 'Prepare'. (3) All in members have received the 'Commit' or 'Abort'. Case (1) is solved by a termination protocol virtually identical to the one in Section 3. Cases (2) and (3) require DM intervention. When a DM that processed an operation for t notices the TM failure, it contacts another member, T', of the logical TM. If T' has not received the 'Prepare', T' attempts to abort t by invoking the Abort transaction. If T' has received the 'Commit' or 'Abort', T' completes the protocol by executing the last step or phase.

An offline TM supports a single function, Rollforward. The TM obtains the CTL, ATL, and PTL stored by some online member of its logical TM, and updates its own lists accordingly.

We now describe TM behavior in response to operations issued by transactions.

- $Begin_t$. The TM assigns transaction t a globally unique transaction identifier. All messages sent by the TM on t's behalf carry this identifier.
- $Read_t(X)$. The TM issues $Read_t(x)$, where x is the copy of X in t's materialization. The data returned by DM x is passed to t. If x rejects the Read, the TM aborts t.
- $Update_t(X)$. Similar to Read.
- $Abort_t$. The TM invokes the Abort transaction to abort t.
- End_t. The TM invokes atomic commit.

6. Failures and Recoveries

When a DM or TM fails or recovers, other parts of the system must react. This section describes system behavior in response to failures and recoveries.

6.1 DM Failures

When DM x fails, the incarnation level will Exclude it. This has an effect on transactions that access x. If transaction t updates logical file X, t cannot commit until x is Excluded. (This is enforced by the Broadcast of End_t to all copies of X; see Section 5.) If t reads x, it will be aborted if x is Excluded before t reaches Phase 2 of commit. (This is enforced by materialization watching, see Section 5.) These conditions are used to avoid the replicated data anomaly illustrated in the Introduction.

6.2 TM Failures

When a TM, T, fails and is excluded, the system may have to abort transactions that were controlled by T. This is governed by the termination protocol described in Section 5.

6.3 DM Recoveries

When a DM x recovers, it is in the offline state and cannot process database operations. DM x moves to the online state by running a variant of Atar et al.'s recovery algorithm [ABG].

Step 1. Set aside a copy of the status database. (This is needed for total failure, see Section 6.5.)

Step 2. The incarnation level Includes x.

Step 3. DM x executes the Rollforward operation to bring itself up-to-date. (See Section 4.) Rollforward requires an online copy of the logical file. If no online copy exists, this is a total failure case and is handled in Section 6.5.

Step 4 DM x sets its substate to online and discards the database saved in Step 1.

If DM x was down for a long time, the Rollforward might take a long time to complete. During this period no transaction that updates logical file X can commit. To shorten this period, DM x can execute Rollforward before being Included. This will bring x 'almost' up-to-date and allow the Rollforward in Step 3 to complete more rapidly.

6.4 TM Recoveries

The algorithm of Section 6.3 works for TM recoveries too, except the Rollforward operation is the one defined in Section 5. That is, it brings the CTL, ATL, and PTL at the recovering TM up-to-date.

6.5 Recovery from DM Total Failure

A DM total failure occurs where all online members of a logical DM have failed. Recovery from DM total failure is similar to recovery from total failure described in Section 3.3.

As DMs recover, they execute the LAST SURVIVORS algorithm of Section 3.3. The algorithm gets its initial values from the status database saved in Step 1 of the DM recovery procedure (see Section 6.3) and restricts all values to members of the logical DM. When the algorithm terminates, it has identified the last surviving DMs. Each DM then resolves any pending transactions. For each pending transaction t, the DM obtains t's status from any member of t's logical TM.

An important special case of DM total failure is the case of nonreplicated data. If x is the only copy of logical file X, every failure of x is a total failure. In this case, the LAST SURVIVORS algorithm terminates immediately, and x need only resolve pending transactions.

6.6 Recovery from TM Total Failure

TM total failures are analogous to the DM case and are handled similarly. When the last survivors are found, pending transactions are resolved by running the commit termination protocol (see Section 5) for each one.

7. Conclusion

Replication is the key factor in making a ddbs more reliable than a centralized dbs, replicated data management and replicated transaction management. A ddbs reliability algorithm is, first and foremost, an expert at handling replication.

Our algorithm makes the following guarantees concerning replicated data.

1. The copies of each logical file behave like a single copy from the standpoint of logical correctness.
2. A transaction can execute provided at least one copy of each logical file it references is available.
3. When a copy of a file recovers it can be reintegrated into the system provided at least one other copy is already available.
4. If all copies of a file fail, the file will become available again when 'enough' of the copies recover.

Our algorithm makes similar guarantees concerning replicated transaction management.

8. References

[ABDG] Alsberg P.A., G.G. Belford, J.D. Day, and E. Grapa. 'Multi-copy Resiliency Techniques,' Distributed Data Management (J.B. Rothnie, P.A. Bernstein, D.W. Shipman, eds.), IEEE, 1978, pp. 128-176.

[ABG] Attar R., P.A. Bernstein, and N. Goodman. 'Site Initialization, Recovery, and Back-up in a Distributed Database System,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 185-202.

[AD] Alsberg, P.A., and J.D. Day. 'A Principle for Resilient Sharing of Distributed Resources,' Proc. 2nd Intl. Conf. Software Eng., Oct. 1976.

[ADEH] Andler, S., I. Ding, K. Eswaran, C. Hauser, W. Kim, J. Mehl, R. Williams. 'System D A Distributed System for Availability,' Proc. 8th VLDB, Sept. 1982, pp. 33-44.

[Ba] Bartlett , J.F. 'A 'NonStop' Operating System,' in The Theory and Practice of Reliable System Design, (Siewiarek and Swarz, eds.), Digital Press, 1982, pp. 453-460.

[BG1] Bernstein, P.A., and N. Goodman. 'Concurrency Control in Distributed Database Systems,' ACM Computing Surveys 13, 2(June 1981), pp. 185-221.

[BG2] Bernstein, P.A., and N. Goodman. 'A Sophisticate's Introduction to Distributed Database Concurrency Control,' Proc. 8th VLDB, Sept. 1982, pp. 62-76.

[BGH] Bernstein, P.A., N. Goodman and V. Hadzilacos. 'Recovery Algorithms for Database Systems,' Proc. 9th IFIPS Congress, Sept. 1983.

[Co] Cooper, E.C. 'Analysis of Distributed Commit Protocols,' Proc. ACM SIGMOD Conf. on Management of Data, ACM, June 1982, pp. 175-183.

[CB] Cheng, W.K. and G.G. Belford. 'The Resiliency of Fully Replicated Distributed Databases,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 23-44.

[CDFLNR] Chan, A., S. Danberg, S. Fox, W-T.K. Lin, A. Nori, and D. Ries. 'Storage and Access Structures to Support a Semantic Data Model,' Proc. 8th VLDB, Sept. 1982, pp. 122-130.

[CFLNR] Chan, A , S. Fox, T.A. Landers, A. Nori, and D. Ries. 'The Implementation of an Integrated Concurrency Control and Recovery Scheme,' Proc. ACM SIGMOD Conf. on Management of Data, June 1982, pp. 184-191.

[Do1] D. Dolev. 'Unanimity in an Unknown and Unreliable Environment,' Proc. 22nd IEEE Symp. on Foundations of Computer Science, IEEE, 1981, pp. 159-168.

[Do2] Dolev, D. 'The Byzantine Generals Strike Again,' J. of Algorithms, 3, 1 (1982).

[DR] Dolev, D. and R. Reischuk. 'Bounds on Information Exchange for Byzantine Agreement,' Proc. 1st ACM SIGACT-SIGOPS Symp. on Principles of Dist. Computing, ACM, Aug. 1982, pp. 132-140.

[DS1] Dolev, D. and H.R. Strong. 'Polynomial Algorithms for Multiple Processor Agreement,' Proc. 14th ACM SIGACT Symp. on Theory of Computing, May 1982, pp. 401-407.

[DS2] Dolev, D. and H.R. Strong. 'Distributed Commit with Bounded Waiting,' Proc 2nd Symp. on Reliability in Distributed Software and Database Systems, IEEE, July 1982.

[DS3] Dolev, D. and H.R. Strong. 'Requirements for Agreement in a Distributed System,' Proc. 2nd Int'l Symp. on Distributed Databases, Berlin, Sept. 1982.

[Ea] Eager., D.L. 'Robust Concurrency Control in a Distributed Database,' Univ. of Toronot TR CSRG #135, Oct. 1981.

[EGLT] Eswaran, K P , J.N. Gray, R.A. Lorie, and I.L. Traiger. 'The Notions of Consistency and Predicate Locks in a Database System,' Commun. ACM, Vol. 19, No 11, Nov 1976, pp. 624-633.

[FFL] Fischer, M J., R. Fowler, and N.A. Lynch. 'A Simply and Efficient Byzantine Generals Algorithm,' Proc. 2nd Symp. on Reliability in Distributed Software and Database Systems, IEEE, July 1982.

[FLP] Fischer, M.J., N A. Lynch, and M.S. Paterson. 'Impossibility of Distributed Consensus with One Faulty Process,' Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Ssytems, ACM, Mar. 1983.

[FM] Fischer, M.J. and A Michael. 'Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network,' Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, ACM, Mar. 1982, pp. 70-75.

[Gi] Gifford, D.K. 'Weighted Voting for Replicated Data,' Proc. 7th Symp. on Operating Systems Principles, ACM, Dec. 1979, pp. 150-159.

[Gr] Gray, J.N. 'Notes on Database Operating Systems,' in Operating Systems an Advanced Course, Springer-Verlag, 1979.

[GMBLL] Gray, J.N., P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzulo, and I. Traiger. 'The Recovery Manager of the System R Database Manager,' ACM Computing Surveys, 13, 2 (June 1981), pp. 223-242.

[HR1] Harder, T., and A. Reuter. 'Principles of Transaction Oriented Database Recovery -- A Taxonomy,' Univ. Kaiserslautern TR 50/82.

[HS] Hammer, M.M., and D.W. Shipman. 'Reliability Mechanisms for SDD-1 A System for Distributed Databases,' ACM Trans. on Database Syst., Vol. 5, No. 5 (Dec. 1980), pp. 431-466

[La1] Lamport, L. 'Time, Clocks, and the Ordering of Events in a Distributed System,' CACM, 21, 7 (July 1978), pp. 558-565

[La2] Lamport, L. 'The Implementation of Reliable Distributed Multiprocess Systems,' Computer Networks, I 2 (1978), pp. 95-114.

[LSP] Lamport, L., R. Shostak, and M. Pease. 'The Byzantine Generals Problem,' ACM Trans. on Programming Languages and Systems, Vol. 4, No. 3 (July 1982), pp. 382-401.

[MPM] Menasce, D.A., G.J. Popek, and R.R. Muntz. 'A Locking Protocol for Resource Coordination in Distributed Databases,' ACM Trans. on Database Syst., Vol. 5, No. 2, (June 1980), pp. 103-138.

[PR] Parker, D.S., and R.A Ramas. 'A Distributed File System Architecture Supporting High Availability,' Proc. 8th VLDB, Sept. 1982, pp. 161-184.

[PSL] Pease, M., R. Shostak, and L. Lamport. 'Reaching Agreement in the Presence of Faults,' JACM, 27, 2 (1980), pp. 228-234.

[Re] Reed., D.P. 'Implementing Atomic Actions,' Proc. 7th ACM Symp. on Operating Systems Principles, ACM, Dec. 1979.

[RSL] Rosenkrantz, D.J., R.E. Stearns, and P.M. Lewis. 'System Level Concurrency Control for Distributed Database Systems,' ACM Trans. on Database Syst.,

[Sh] Shipman, D.W. 'The Functional Data Model and the Data Language DAPLEX,' ACM Trans. on Database Syst., Vol. 6, No. 1, (Mar. 1981), pp. 140-173.

[Sk1] Skeen, D. 'Nonblocking Commit Protocols,' Proc. 1982 ACM-SIGMOD Conf. on Management of Data, ACM, pp. 133-147.

[Sk2] Skeen, D. 'A Quorum Based Commit Protocol,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 69-80.

[Sk3] Skeen, D. 'Determining the Last Process to Fail,' Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, ACM, Mar. 1983.

[SkSt] Skeen, D., and M. Stonebraker. 'A Formal Model of Crash Recovery in a Distributed System,' Proc. 5th Berkeley Workshop, 1981, pp. 129-142.

[St] Strom, B.I. 'Consistency of Redundant Databases in a Weakly Coupled Distributed Computer Conferencing System,' Proc. 5th Berkeley Workshop, 1981, pp. 143-153.

[TGGL] Traiger, I.L., J. Gray, C.A. Galtier, and B.G. Lindsay. 'Transactions and Consistency in Distributed Database Systems,' ACM Trans. on Database Systems, Vol. 7, No. 3, (Sept. 1982), pp. 323-342.

[Vl] Verhofstad, J.M.S. 'Recovery Techniques for Database Systems,' ACM Computing Surveys, 10, 2 (1978), pp. 167-196.

[Wa] Walter, B. 'A Robust and Efficient Protocol for Checking the Availability of Remote Sites,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 45-68.