# Dynamic Data Distribution ($D^3$) in a Shared-Nothing Multiprocessor Data Store

Donald D. Chamberlin and Frank B. Schmuck
IBM Research Division
Almaden Research Center

## Abstract

Redundant storage of data can make a multiprocessor database system tolerant of single processor failures. The $D^3$ algorithm described here enables such a system to dynamically reorganize after a failure in order to protect itself against further failures, without interruption of service. The algorithm also permits addition or removal of a processor, and data migration for load balancing, without interruption of service.

## 1 Introduction

Multiprocessor database systems provide a number of advantages over uniprocessors, including increased capacity and modular growth. An important class of multiprocessor systems is based on the "shared-nothing" approach, in which several processors, each with its own memory and disk storage, manage a shared database by exchanging messages. In such a system, fault-tolerance can be improved by replication of data. If each block of data is replicated on two processors, the system can survive any single failure without loss of data or interruption of service. However, after a single failure, some of the data in the system no longer has a second copy, and so the system is not protected against subsequent failures. This problem is typically dealt with in one of the following ways:



**Proceedings of the 18th VLDB Conference Vancouver, British Columbia, Canada 1992**

1. The system continues running, operating on the surviving copy of the missing data. All changes that would have been applied to the data on the failed processor are recorded in a log. When the failed processor rejoins the system, it applies the log to its data in order to "catch up" to a state consistent with the other processors. The disadvantages of this approach are that the log can become quite lengthy, and the system is unprotected until the failed processor is back online.

2. The system is halted at some planned time, and its data is reorganized in such a way that all data is once again replicated and the load on the surviving processors is properly balanced. The disadvantage of this approach is that it requires an interruption of service.

This paper proposes a new approach to the reorganization of data after a single failure, called Dynamic Data Distribution ($D^3$). In the $D^3$ approach, the surviving copies of lost data blocks automatically replicate themselves and migrate to new processors without operator intervention, and the system returns itself to a "protected" state without any interruption of service. The approach also permits processors to be added to or deleted from the system without an interruption of service. The $D^3$ approach is well-suited for transaction-processing environments that have at least one of the following properties:

1. Any interruption of service is extremely costly or undesirable, including "planned" shutdowns to reconfigure the system after a failure, or for load-balancing, or to add capacity to the system.

2. The system is required to provide reliable service without operator intervention over a long period of time which may span multiple processor failures.

Immediately after a processor failure, there is an interval of time during which replication of data takes

place. During this interval, transaction processing continues but an additional processor failure would require the system to stop and recover data from a log. If a higher degree of fault tolerance is needed, the $D^3$ algorithm easily generalizes to maintain three or more copies of each data block, thus tolerating two or more simultaneous failures without interruption of service.

The $D^3$ algorithm assumes that a cluster of processors ("nodes") is acting as a network server, accepting and processing database transactions. The objectives of the algorithm are as follows:

1. **Single image:** To a client, the cluster of nodes should appear like a single machine. The distribution of data across nodes should be invisible to clients.

2. **Modular growth:** The system should permit new nodes to be added, increasing throughput and capacity without any interruption in service.

3. **Fault tolerance:** If a single node fails or is removed from the cluster, throughput may go down but there should be no loss of data or interruption of service.

4. **Self-repair:** After the failure of a single node, the cluster may be temporarily in an "unprotected" state in which the failure of another node would require interruption of service and offline recovery. The system should redistribute its data to return to a "protected" state quickly and automatically. The system should know when its state is "protected".

5. **Safe restart:** After an interruption of service due to multiple failures (e.g., a power loss) the system should be able to restart with minimal operator intervention.

6. **Transaction semantics:** All database access should consist of a series of transactions, with the usual guarantees of isolation, atomicity, and recoverability.

7. **Scalability:** There should be no bottleneck that limits the addition of new nodes to the cluster.

8. **Symmetry:** No node should be more critical to continued operation than any other.

9. **Dynamic load balancing:** The system should automatically redistribute data to balance the load on the nodes for optimum performance. This should be done without interruption of service.

The rest of this paper is organized as follows: Section 2 states the assumptions on which the algorithm is based. Section 3 gives an overview of the algorithm and a discussion of the the problems unique to the Dynamic Data Distribution approach. Sections 4–9 describe the steps of the algorithm in more detail. Sections 10 and 11 give an outline for a proof of correctness and suggest possible extensions to the algorithm. The paper concludes with a comparison to related work and references.

# 2 Assumptions

The $D^3$ algorithm is based on the following assumptions:

1. Messages between nodes in the cluster are cheap, fast, and reasonably reliable (lost messages are tolerated but cause a degradation of performance similar to a node failure.)

2. Nodes fail by becoming unresponsive (they stop replying to messages). If a node does respond to a message, we assume the reply is correct.

3. Each message transmitted between nodes results in a reply or acknowledgement. The sending node may engage in other work while waiting for the reply to a message. If a reply is not received within a designated time, the sending node assumes that the receiving node has failed.

4. Each node has a unique NodeID, and there is a natural ordering among NodeID's.

5. The database consists of a set of *records*, each of which has a unique name called its record-identifier (RecID). Groups of records (or potential records, represented by unused RecID's) are statically assigned to *blocks*, each of which has a unique name called its block-identifier (BlockID). There is a simple mapping from RecID to BlockID (e.g., BlockID could be a prefix of RecID). A record is the unit of user access to data, and the unit of locking. A block is the unit of data distribution and migration between nodes. No assumption is made about the number of records per block. The motivation for assigning multiple records to a block is to keep the lockable unit small (to increase concurrency) and the migratable unit larger (to reduce message traffic).

6. Each individual node implements local transaction semantics, providing the following guarantees:

(a) **Local isolation:** Each node prevents multiple transactions from interfering with each other by acquiring locks on modified records. Higher degrees of isolation could also be supported (e.g., locks could be acquired on records when they are read as well as when they are modified.)

(b) **Local atomicity:** Each node ensures that all local updates made by a given transaction are committed or rolled back as an atomic act.

(c) **Local recoverability:** If a node fails, an off-line recovery procedure is available whereby the data in that node can be recovered to a transaction-consistent state.

We assume that the nodes participating in the $D^3$ algorithm implement these local guarantees using a standard technique [Gray 81, Mohan 89]. The $D^3$ algorithm builds on the local guarantees to ensure isolation, atomicity, and recoverability for the system as a whole.

# 3 Overview

Initially, each block of data is stored in two different nodes, called the *storage sites* for the block. There is a *partition function* (PF) which maps a BlockID to its storage sites. The PF can be used to find the locations of an existing block, or to select two sites for a new block. When a transaction updates a record the update is sent to both nodes that store the block containing the record. Record locking and a two-phase distributed commit protocol are used to ensure transaction isolation and atomicity.

If there were no failures and no need for load balancing, the initial data distribution would never change. Load balancing, however, requires blocks to migrate between nodes, and a failure requires new copies to be created of those blocks for which the failed node was one of the two storage sites (in order to protect against subsequent failures.) For this purpose, one of the nodes, the one with the lowest NodeID, is designated the *Coordinator*. The Coordinator gathers load statistics and information about failures from the other nodes and decides when a new partition function needs to be created to balance the load, to reconfigure after a node has failed, or to integrate a new node into the system. The details of the new PF are left to the Coordinator, but in order to minimize movement of data, each new PF should resemble the previous PF as closely as possible.

The Coordinator transmits each new PF to all the other nodes. When a node receives a new PF it begins scanning through all its blocks, using background cycles. For each block, the node applies the new PF to determine the new storage sites for the block. If the new sites differ from the current storage sites, the node sends a copy of the block to the new sites and deletes the local copy (unless the node itself is one of the new sites.) The protocol for distributing a new PF and for migrating blocks is described in more detail below. When a node finds that all its blocks are stored according to the most recent PF, it declares itself "Up-To-Date" and stops scanning until a new PF is distributed.

When all nodes are Up-To-Date the reconfiguration is complete, and there are again two copies of all data blocks, including the ones that were stored at the failed node. At this time the system is protected against a subsequent failure. Note that the $D^3$ algorithm does not maintain a separate log of "missed updates" that a failed node could use to bring itself up-to-date after it has recovered from its failure. Instead, a failed node that has been repaired is treated like a new node: it is assigned a new NodeID, the Coordinator creates a new PF that includes the node, and data begins migrating back to the repaired node.

During reconfiguration after a failure the system is *unprotected*: if a second failure occurs before the reconfiguration is complete, some data blocks may be unavailable. In this case the Coordinator will shut down the system and must wait until the second failed node has come back online and has recovered its data from its local log.

In the absence of block migration, i.e., without failures or load balancing, the distributed commit protocol alone would be sufficient to guarantee consistency between the two copies of a block. Block migration and reconfiguration after a failure introduce two problems that are unique to the Dynamic Data Distribution approach:

1. **Finding data blocks:** After a new PF has been distributed and blocks begin to migrate, there are no longer only two nodes at which a copy of a block may be stored. In fact, during the transition to a new PF, there may be more than two copies of a block. When a record is updated, the algorithm needs to ensure that all copies of the block containing the record are found and updated, and that no block is missed while it is "in transit" from one node to another. Similarly, when processing a read request, all potential storage sites need to be contacted before deciding that a record and the block to which the record belongs do not yet exist.

2. **Detecting "stale" data:** As described earlier, a node that comes back online after a failure discards all its data and rejoins the system under a new NodeID. This is done because, due to updates the node missed while it was down, its data may have become *stale*. During recovery from multiple failures (e.g., when the system restarts after a power failure), however, no data should be discarded, except for data on nodes that had failed even earlier (e.g., before the power loss). Hence, a node that restarts after a failure must be able to reliably decide whether its data is up-to-date and should be retained or potentially stale and should be discarded.

Below we describe how these problems are solved in the $D^3$ algorithm.

**Finding Blocks:** Every node keeps a list of all *Active PF's*, i.e., all PF's that still control the storage of one or more blocks at some node. Each node informs the Coordinator when a PF no longer has any blocks stored according to it at that node. This allows the Coordinator to determine when a PF is no longer active. The Coordinator will then send a message to all nodes informing them that the PF can be removed from the list of active PF's.

In order to find all copies of a block, a node applies all PF's in its list of Active PF's to the BlockID, yielding the NodeID's of all nodes that might have a copy of the block. This list of NodeID's is called the *trail* of the block. The nodes on the trail are ordered from oldest PF to newest PF, and duplicates are not eliminated. When a transaction running at Node $N$ reads or updates a record, the node follows the trail of the block containing the record, sending a message to each node on the trail until the desired block has been found (in case of a read request) or all copies have been updated (in case of an update request).

A two-phase protocol is used by the Coordinator to distribute a new PF to all nodes: First, the Coordinator sends a *Prepare for new PF* message, describing the new PF, to every node. The Coordinator waits for an acknowledgement from all nodes and then sends an *Activate New PF* message to each node. Block migration and creation are suspended during the distribution of a new PF, i.e., after receiving a new PF a node waits for the *Activate New PF* message before migrating or creating any new blocks. This ensures that no blocks are stored according to a new PF until all nodes have received that PF.

When a block migrates from a node $N_1$ to $N_2$, $N_1$ places a *Migration Lock* lock on the block until $N_2$ has acknowledged the receipt of the block. If node $N_1$ receives a read or update request from node $N$ affecting a migrating block, $N_1$ will wait for the migration lock to be released before replying "block not here" to $N$. Thus $N$ will not send its request to $N_2$ until the block has been received by $N_2$. (Since nodes on the trail are ordered from oldest PF to newest, $N_1$ must precede $N_2$ in the trail.) This ensures that $N$ does not miss the block while it is in transit from $N_1$ to $N_2$.

**Detecting Stale Data:** To explain how a node decides whether its data might be stale, we need to describe in more detail how the Coordinator decides whether the system can continue to operate after a failure or needs to shut down. As described above, the Coordinator keeps track of all PF's that are currently active. We call the set of nodes on which data blocks are stored under a given PF the *Participation Set* of the PF. If the Participation Set of any active PF contains two or more nodes that have failed since the PF was created, then it is possible that some data block is unavailable because its only two copies were stored at failed nodes. If, on the other hand, the Participation Set of each active PF contains at most one failed node, then every data block must have at least one copy stored at a node that is still available.

Data at a failed node becomes stale when a transaction that updates blocks that were stored at the failed node is committed. During the distribution of a new PF commit processing is suspended, i.e., after receiving a new PF from the Coordinator, a node will not process any commit requests until it has received the *Activate new PF message*. Therefore, data at a failed node $N$ cannot become stale until a PF with a Participation Set excluding $N$ has been activated at at least one node. Hence when $N$ recovers it can decide whether its data might be stale by determining whether there is any node that has activated a PF from which $N$ was excluded.

Notice that it is possible that new nodes were added to the system since $N$ had failed. However, as the following argument will show, it is sufficient for $N$ to contact only nodes that were active at the time $N$ failed, i.e., those nodes that are in the Participation Set of the most recent PF $f$ known to $N$. At the time of $N$'s failure $f$ was in the list of active PF's at all nodes. Therefore, the next PF created by the Coordinator must include all nodes in the Participation Set of $f$, except for $N$; otherwise, by the rules give above, the Coordinator would have shut down the system instead. Hence, because of the two-phase protocol for PF distribution, a PF excluding $N$ will not be activated at any node until all other nodes in the Participation Set of $f$ have received the new PF.

# 4 Primary Messages

In this and the following sections we describe the steps of the $D^3$ algorithm in more complete detail.

Incoming requests from applications to the database system are grouped into transactions. Each transaction is assigned to a particular node, called its *T-node*. A transaction may be thought of as sending *primary messages* to the data manager in its T-node. These primary messages define the client interface of the reliable storage implemented by the $D^3$ algorithm. While processing primary messages on behalf of a transaction, a T-node may send additional messages, called *secondary messages*, to other nodes.

When a T-node receives a primary message to *Get*, *Store*, or *Delete* a record with a given RecID, it maps the RecID into a BlockID and computes the *trail* for that block. The T-node then follows the trail of the desired block, sending secondary messages to each node on the trail, including itself. If any of these secondary messages results in a lock conflict, the receiving node returns a "Lock Wait" response, causing the T-node to suspend the transaction until it it is notified that the lock has been released. After reaching the end of the trail, the T-node checks to see if the trail has been extended by the addition of a new PF; if so, it continues to send secondary messages to the nodes on the extended trail until it reaches the end and verifies that the trail has not been extended.

We can see the reason for following the trail from oldest PF to newest PF, and for not eliminating duplicate nodes on the trail, if we visualize how a block may be migrating at the same time that a T-node is searching for it. The migration will always proceed from an older to a newer PF, and a lock will be held on the block at the migrate-from site until the block is safely stored at the migrate-to site. Therefore, a T-node that follows the trail of the block from oldest to newest PF, including duplicates, is guaranteed to "catch up" to a migrating block. By following the complete trail, the T-node can ensure that an operation such as update or delete is applied to all existing copies of a record.

The primary messages, and the secondary messages that arise from them, are as follows:

- **Begin Transaction:** The node receiving this message assigns a TranID to the transaction and becomes its T-node. The TranID is included in all secondary messages sent on behalf of this transaction. During the life of the transaction, the T-node keeps a list of all nodes at which data has been modified by this transaction (including nodes that failed before responding to a request to modify data).

- **Get (TranID, RecID):** The intent is to retrieve the content of the record with the given RecID. The T-node sends a *Simple Get* secondary message to each node on the trail, in order. *Simple Get* causes the receiving node to search its store and to return one of the following outcomes: (a) the desired record; (b) a "No Such Record" code if the block is present but the record does not exist; or (c) a "Block Not Here" code. As soon as the T-node receives outcome (a) or (b) from some node on the trail, it can return to the application without following the remainder of the trail.

- **Store (TranID, RecID, Data):** The intent is to replace the record with the given RecID, or, if this record does not exist, to create it. The T-node sends a *Simple Update* secondary message to each node on the trail, in order. *Simple Update* causes the receiving node to search its store for the block containing RecID. If the block is found and the record exists, the receiving node updates it (unless it is locked) and returns a "Success" code. If the block is found and the record does not exist, the receiving node creates a new record with the given RecID and data (unless its absence is locked) and returns a "Success" code. If the block is not found, the receiving node returns "Block Not Here".

  When the T-node arrives at the end of the trail, if one or more nodes responded to a *Simple Update* message with a "Success" code, the T-node adds these nodes to the modified-nodes-list for the current transaction, and returns a success-code to the application. However, if no "Success" response was received, the given block does not yet exist, and must be created. In this case, the T-node determines the two storage sites for the given block according to the Current PF, sends a *Simple Create* secondary message to each of them, and adds them to the modified-nodes-list. The *Simple Create* messages cause the receiving nodes to create a new block, insert a new record with the given RecID and data, and to lock the new record on behalf of the requesting transaction.[1]

- **Delete (TranID, RecID):** The intent is to delete the record with the given RecID. The T-node sends a *Simple Delete* secondary message to

[1] Under certain circumstances, a node receiving a *Simple Create* request from some T-node A may discover that the given block already exists, because a transaction running at another T-node B has created it in the interval since "Block Not Here" was reported to T-node A. In this case, the receiving node simply creates the new record in the existing block, waiting for a lock to be released if necessary.

each node on the trail, in order. *Simple Delete* causes the receiving node to search its store, deleting the given record if it exists and is not locked, and to return a code indicating "Deleted," "No Such Record," or "Block Not Here." When the end of the trail is reached, if one or more nodes responded "Deleted" to the *Simple Delete* request, the T-node adds these nodes to the modified-nodes-list for the current transaction.

- **Commit Transaction (TranID):** The T-node commits a transaction by means of a standard two-phase commit protocol [Gray 79] involving all the nodes on the modified-node-list for the transaction. A *Prepare to Commit Local Transaction* secondary message is sent to all participating nodes (including the T-node itself), causing each participating node to force the necessary log pages to disk and to confirm that the transaction can be committed. When all participating nodes have responded affirmatively, the T-node sends a *Commit Local Transaction* message to all participating nodes (including itself), causing each node to log the commit action, release all locks held on behalf of the transaction, and send *Lock Released* messages to the nodes that are waiting for these locks. As a result of the lock releases, some blocks may become free to migrate.

  If any node on the modified-node-list for the given transaction has failed, the T-node does not commit the transaction until a new PF that excludes the failed node has been distributed and activated. As explained above, this is necessary to ensure that that if the failed node "wakes up", it will be able to detect that its data is stale.

  After each participating node has locally committed a transaction, the node continues to remember the transaction-ID and status as a committed transaction. This information may be needed if the transaction's T-node fails before all participating nodes have locally committed. In this case, the Coordinator will take over as T-node for the transaction, and will determine the status of the transaction by polling the other nodes. The ID of a committed transaction is retained by each participating node until the T-node sends it a *Flush Transaction ID* message, confirming that all participating nodes have committed or aborted the transaction (these *Flush* messages can be batched together and distributed periodically, or piggybacked on other messages.)

- **Rollback Transaction (TranID):** When a *Rollback Transaction* is received from the application, or when any node responds negatively to the two-phase commit protocol, the T-node sends a *Rollback Local Transaction* message to all participating nodes, and returns a *Rollback* code to the application. The *Rollback Local Transaction* message causes each node to undo all data modifications made by the given transaction, log the rollback action, release all locks held by the transaction and send *Lock Released* messages to the nodes that are waiting for them. Nodes that perform local rollbacks also retain the transaction-ID and status of the rolled-back transaction until a *Flush* message is received from its T-node.

  A T-node also rolls back a transaction if it has been suspended for more than a designated time limit, presuming the transaction to be deadlocked. When the application receives a Rollback code, it can choose to resubmit the transaction.

# 5 Node Failures

Whenever a node fails to respond to a message within a designated interval, it is presumed to have failed. The node that detects the failure is called the *detecting node*. The detecting node sends a *Failed Node (NodeID)* message to the Coordinator (which may be the detecting node itself). However, if the failed node is the Coordinator, the detecting node finds the Coordinator's successor, which is the next-largest non-failed NodeID after the Coordinator, and sends this successor node a message, *You're the New Coordinator (NodeID failed)*. Having performed the proper notification, the detecting node can now go on about its business. The messages generated by the detecting node are processed as follows:

- **You're the New Coordinator (NodeID failed):** This message may be received multiple times, as many nodes discover that the old Coordinator has failed (the second and subsequent notifications are ignored.) The node receiving this message must prepare itself to take over the Coordinator's job. The new Coordinator, like all nodes, has a complete list of Active PF's; but to serve as Coordinator, it needs to know which PF's are active at each node. The new Coordinator reconstructs this data by sending a message, *Report Status*, to all nodes in the system, to which each node responds by listing its locally active PF's. The newly elected node now has all the information it needs to serve as Coordinator, and it can proceed as if it had just received the message, *Failed Node (NodeID)*, containing the NodeID of the old Coordinator.

- **Failed Node (NodeID):** This message is received by the Coordinator, whose first responsibility is to determine whether the system can continue to operate. If any PF on the list of Active PF's has a Participation Set that includes more than one failed node, some data blocks may no longer be available, and the system must stop and perform a log-based recovery. In this case, the Coordinator sends a *Shutdown* message to all nodes (including itself), and prints a message to the operator: "NodeID failed, log recovery required."

  If all Active PF's have at least two operating nodes and no more than one failed node in their Participation Sets, the system can continue to operate. In this case, the Coordinator prints an operator message, "NodeID failed, no recovery needed," informing the operator that the given node can be switched off and removed from the system. The system will continue operating with no interruption of service. When the failed node is repaired, its disks can be erased and it can rejoin the system as a new node with a new NodeID.

  After a node failure that does not require log recovery, the Coordinator protects the system against additional failures by making up a new PF that includes in its Participation Set only those nodes that are still in operation, and propagating this new PF to all the surviving nodes by means of a two-phase protocol, as follows:

  1. First, the Coordinator sends a *Prepare for new PF* message, describing the new PF, to every node. Each node accepts the new PF, retains the previous PF in its active-PF list, and goes into a PF-TRANSITION state in which it temporarily refuses to process commit request and to store any new blocks. Each node also responds with a list of all the transactions it knows about for which the failed node is the T-node, and the status of each (in progress, aborted, prepared to commit, or committed.)
  2. When the Coordinator has confirmed that all nodes have received the new PF and are in PF-TRANSITION state, it sends an *Activate New PF* message to each node, causing it to resume storing blocks according to the new PF and to reactivate any of its own transactions that are waiting for locks in the failed node. The Coordinator also "takes over" as T-node for all transactions whose T-node was the failed node. If such a transaction is locally committed at any node, the Coordinator instructs all nodes to commit the transaction; otherwise it instructs all nodes to abort the transaction.

If the Coordinator fails during the transition to a new PF, only a subset of all nodes may receive the the *Prepare for new PF* message. However, the new Coordinator will generate an even newer PF, and all surviving nodes must receive this new PF before block migration is resumed. This ensures that a block is never stored according to a partially distributed PF.

The mechanism described above treats the PF as a small database that is replicated at every site and kept consistent by a two-phase distribution protocol. This ensures that all processes see a consistent PF, regardless of the node(s) at which they execute. However, even if the PF is changed at all sites as an atomic act, the stored data requires some time to migrate to new sites according to the new PF. This is the reason for the concept of a *trail*, introduced earlier, and for the data migration algorithm described below.

Upon receiving an *Activate New PF* message, each node begins (or resumes) a scan through all the blocks in its store, looking for blocks that are not stored according to the latest PF. A mark containing the new PF-id is placed at the starting point of this scan; when the scan cycles back to this mark, the node will know that no PF's earlier than this PF-id remain active at this node. When the scan encounters a block that is stored according to an old PF, the block is sent to its two storage sites under the Current PF. However, if the block that needs to migrate contains any locked records, it is placed on a migration-pending list, where it remains until the locks are released.

When a block is free to migrate, the node holding the block locks it (preventing T-nodes that are searching for records in this block from missing the block while it is in transit), and sends the block in a *Migrate* message to its two new storage sites (if the current node is one of the new sites, it simply retains a copy of the block.) A node receiving a *Migrate* message searches its own store for an existing block with the given BlockID. If no such block is found, the receiving node accepts the new block into its data store. If an existing block is found with the same BlockID, the receiving node simply keeps the existing block and ignores the incoming block (returning a normal reply to the *Migrate* message.) This condition may occur when both old storage sites send a copy of the block to both new storage sites. These multiple messages are not harmful; in fact, they are the mechanism by which surviving copies of blocks replicate themselves after a failure. The migrating block contains no uncommitted updates—otherwise the modified records would be

locked and the block would not be migrating. The copy of the block that already exists at the receiving node may contain some updates that are not yet present in the migrating block; therefore, the receiving node keeps the existing copy of the block rather than the migrating copy.

When the origin node of a migrating block has received confirming replies from the two new storage sites, it unlocks the block and deletes it from its store (unless it is also one of the new storage sites.)

In the process of scanning stored records, if a mark is encountered containing a PF-id, the scanning node knows that all earlier PF's are no longer active at this node. The node reports this fact to the Coordinator in an *Inactive PF* message. If only the Current PF remains active at this node, the node suspends its background scanning process.

The Coordinator maintains a list of the PF's that are active at each node. When the Coordinator discovers that a given PF is no longer active at any node, it sends a *Delete Old PF* message to all nodes, causing them to delete the given PF from the list of Active PF's. If the Coordinator sees that there are no Active PF's other than the current one, it prints a message to the operator: "The system is now protected." The system can now sustain another node failure without interrupting service.

# 6 Communication Failures

We can now examine the effects on the system of unreliable communication between nodes. Like node failures, communication failures can lead to system reorganization or shutdown but will not cause data to be lost or database integrity to be compromised. If a message to (or reply from) node $N$ is lost, the sender of the message will conclude that node $N$ has failed and will notify the Coordinator. To minimize system reorganizations due to temporary communication outages, the sending node can resend a message one or more times before declaring the receiving node to have failed.

As described earlier, when the Coordinator is informed that node $N$ has failed, it distributes a new PF that excludes node $N$, and instructs the operator to take node $N$ offline. The other nodes stop communicating with node $N$, and replicate its data. Node $N$ will soon discover that it has been shut down, since all nodes respond with a "shutdown" code to messages from nodes that are not in the Current PF. When communications are re-established, node $N$ can rejoin the system as an empty node with a new NodeID.

Multiple communication failures have the same results as multiple node failures, possibly causing the system to shut down and perform log-based recovery. If communication failures cause the system to be partitioned into two parts, neither partition will have enough nodes to continue processing (except in the degenerate case where one partition contains a single node.)

Long delays in responding to messages, possibly due to queuing delays as the system becomes overloaded, may also be interpreted as node failures and may lead to system reorganization or shutdown. This possibility can be minimized by adjusting the length of time that a node waits for a reply to a message before deciding that the target node has failed.

# 7 Multiple Failures

If a node fails while the system is in an unprotected state, it may be necessary to stop the system and recover data from the log of the failed node. As described earlier, this condition is discovered when the Coordinator node receives a *Failed Node* message, and finds that some Active PF now has a Participation Set that includes more than one failed node. The Coordinator then sends a *Shutdown* message to all surviving nodes in the system, and prints a message to the operator: "NodeID failed, log recovery required." When each surviving node receives the *Shutdown* message, it performs local rollback processing for all unprepared transactions (undoing their data changes), and releases all locks. It then enters a SHUTDOWN state, in which it no longer accepts database access requests.

Another important scenario is the case in which multiple nodes fail simultaneously, perhaps due to a common cause such as failure of a power supply. As in the "second individual failure" scenario above, the surviving nodes (if any) appoint a new Coordinator (if necessary) and enter the SHUTDOWN state.

Operator intervention is needed to resume processing after the system has shut down. As each failed node becomes ready for service, the operator allows it to rejoin the network in the SHUTDOWN state. The disk storage of such a newly-reactivated node may not be in a transaction-consistent state, since the node may have failed while transactions were in progress. In addition, the node's data may be "stale", since this node may have been the first to fail and the other nodes may have continued processing, allowing more transactions to commit.

When a sufficient number of nodes are operational for the system to resume processing,[2] the operator sends a *Reset* command to all operational nodes, caus-

[2] At least $n-1$ nodes, where $n$ is the number of active nodes when the system was last in a protected state.

ing each node to decide whether its data is "stale". When a given node $N$ receives a *Reset* command, it polls all the nodes in its most recent PF, asking each node to respond with the latest PF that it knows about. If any node responds with an activated PF that does not include $N$, node $N$ knows that its data is stale and must be discarded. It prints a message instructing the operator to erase the disks of node $N$, give it a new NodeID higher than any existing node, and let it rejoin the system as a new, empty node.

If, on the other hand, the poll does not reveal any active PF that excludes $N$, node $N$ knows that its data is not stale and should be retained. It proceeds by using its local log to abort any unprepared transactions, and remains in the SHUTDOWN state, waiting for further instructions. The fate of transactions whose local status is "prepared" will be propagated in a later message from the Coordinator node.

When all recovering nodes have processed a *Reset* command, the system is ready to resume operation. All nodes are in the SHUTDOWN state. The operator gives a *Resume* command to the node with the lowest NodeID among all the nodes that have recovered. This node is the Coordinator, and it takes responsibility for putting the system back in operation. First, the Coordinator polls all the nodes in the current Participation Set, asking each node to reply with a list of all PF's that are locally active, and a list of all known transactions that have not yet been flushed, and the local status of each. After the poll is completed, the Coordinator takes the following action:

- If more than one node in the Participation Set of any Active PF fails to respond, the Coordinator prints a message, "Not enough machines to resume operation", and remains in the SHUTDOWN state.

- If all nodes in the Participation Sets of all Active PF's respond, the Coordinator sends a *Continue* message to each node. The system is now back in operation and protected against a subsequent failure. The *Continue* message propagates information that the Coordinator has collected about the status of pending transactions, causing transactions that have been locally committed at some node to be committed at all nodes, just as in the case of an *Activate New PF* message.

- If exactly one node in the Participation Set of some Active PF fails to respond, the Coordinator will restart the system in an unprotected state. If that PF was *not* the Current PF, the Coordinator sends a *Continue* message to each node, propagating transaction-status information as above. Otherwise, it makes up a new PF with a Participation Set that excludes the node that failed to respond, and distributes it (by the two-phase protocol of *Prepare for New PF* and *Activate New PF* messages) to all the nodes in its Participation Set. The system is now back in operation.

Upon receiving a *Continue* or *Activate New PF* message, the participating nodes resume normal message processing. Nodes that have more than one locally-active PF resume the background scan for block migration.

# 8 System Reconfiguration

The algorithm described here gives the system operator great flexibility to reconfigure the system by adding or deleting nodes without interruption of service.

To start up the system initially, the operator gives each participating node an *Initialize* command which informs it of its NodeID and the NodeID's of other participating nodes, and places it in the SHUTDOWN state. The operator then gives a *Resume* command to the node with lowest NodeID, causing it to assume the role of Coordinator, generate a PF, and distribute it to all the other nodes. The system is then in operation and ready to be loaded with data.

To add a node to the system while it is running, the operator gives the new node an *Initialize* command, assigning it a NodeID higher than any existing node. The operator then gives the Coordinator an *Add Node* command, informing it of the NodeID of the new node. The Coordinator responds by sending the new node an *Active PF's* message, informing it of the list of Active PF's. The Coordinator then makes up a new PF that includes the new node and distributes it to all nodes, using *Prepare for New PF* and *Activate New PF* messages. In time, data will migrate into the new node based on the new PF. Service is not interrupted, and the protection-state of the system is not degraded during this process.

To delete a node from a running system, the operator gives a *Delete Node* command to the Coordinator, informing it of the NodeID to be deleted. The Coordinator prints an operator message, "Wait for confirmation," and then makes up and distributes a new PF with a Participation Set that does not include the node to be deleted. The Coordinator then waits until the various nodes report that older PF's are no longer active. When the Coordinator finds that the only PF's remaining active are ones that exclude the node to be deleted, it prints an operator message "NodeID may now be taken offline". All the data in the indicated node has now migrated elsewhere, and the node may

be removed from the system. Service is not interrupted, and the protection-state of the system is not degraded during this process.

In addition to the explicit actions taken by the system operator, the Coordinator node can automatically generate a new PF periodically, to rebalance the load on the various nodes. First, the Coordinator measures the average load on each node by sending a *Report Load Statistics* message to all nodes in the system. Then, if the Coordinator decides that data needs to be redistributed for load-balancing purposes, it makes up a new PF and distributes it by means of *Prepare for New PF* and *Activate New PF* messages. This process does not interrupt the system or degrade its protection-state.

# 9 Notes on Performance and Availability

Although at any given time one node is designated as Coordinator, this node does not play a central role that might limit either the performance or the availability of the system. Any node is capable of becoming the Coordinator at any time. The Coordinator plays no special role in the processing of normal messages; its role differs from that of the other nodes only in the event of a node failure.

In normal operation, the system will be in a protected state, and all data will be stored according to the Current PF; thus incoming *Get*, *Store*, and *Delete* messages can be forwarded immediately to the node(s) that hold the relevant data, at a cost of one message for *Get* and two messages for *Store* and *Delete*. Even after a failure, the "trail" of a record is not very long. At worst, a node failure may occur shortly after a new load-balancing PF was introduced, resulting in three Active PF's: the original one, the load-balancing one, and the one resulting from the node failure.

The process of scanning for misplaced data and migrating it to the proper node causes CPU load and message traffic. This scanning process takes place only in the aftermath of a node failure or load rebalance, and uses low-priority "background" cycles. When the migration of data is complete, the scanning process stops. The $D^3$ approach is designed for environments in which the overhead of migrating data is justified by the importance of protecting the system against interruptions due to multiple failures.

A *migration priority switch* could be added to the $D^3$ algorithm, permitting dynamic control over the tradeoff between transaction throughput and quick recovery to a protected state after a failure. These goals are in conflict because a block cannot migrate from one node to another while one of its records is locked. If there are many records per block and a high rate of record updates, this may lead to delays in migration of blocks. Setting a migration priority switch would prevent transactions from acquiring new locks on records in a block that is waiting to migrate. When the existing locks in the block are released, the block is free to migrate. This policy would speed up the process of recovering from a failure, but would increase the probability of transactions waiting for a lock.

# 10 Proof of Correctness

A proof of correctness of the $D^3$ algorithm has been constructed by showing that, for any execution of $D^3$, there exists a valid execution of a standard read-one/write-many algorithm with a two-phase distributed commit protocol [Bern 83] that produces the same results. We refer to the standard algorithm as $S^1$. For a given execution of $D^3$, we define the *historic trail* of a block as the set of all nodes that have stored the block at some time during the execution. We map an execution of $D^3$ into an execution of $S^1$ by assuming that, in the $S^1$ execution, each block is statically assigned to *all* nodes on its historic trail. Nodes never fail in the hypothetical $S^1$ execution, so no messages relating to node failure or data migration exist in $S^1$. Each $D^3$ message that modifies the content of a block (*Simple Create, Simple Delete,* or *Simple Update*) is mapped into a set of equivalent $S^1$ messages to *all* the storage sites of the block (all the nodes on its historic trail). Similarly, commit protocol messages (*Prepare to Commit, Commit,* and *Rollback Local Transaction*) in the $D^3$ execution are propagated, in the $S^1$ execution, to *all* nodes on the historic trail of any block modified by the transaction.

The proof shows that the mapping described above generates a valid execution of $S^1$ that is equivalent to a given execution of $D^3$. This is done by showing that all messages preserve the following invariant: for each node $N$, all data blocks that exist at $N$ in both executions have the same content in both executions (however, some blocks may be present at node $N$ in the $S^1$ execution that are not present in the $D^3$ execution.)

Clearly the $S^1$ approach described here is not a practical alternative for a real system, since it involves massive replication of every block and hypothetical nodes that never fail. It is presented simply to show that the behavior of $D^3$ is equivalent to the behavior of an algorithm known to be correct.

The details of the proof, and a more detailed description of the processing of $D^3$ messages, are given in [Cham 92].

# 11 Possible Extensions

The $D^3$ algorithm is designed primarily for processing transactions that access relatively small sets of records. The algorithm provides parallelism between transactions, permitting a high overall transaction rate. An interesting extension would be to consider parallelism within a single transaction. For example, a transaction might need to update all records satisfying some criterion, and this work might be distributed in the form of a set-oriented request to all nodes at which relevant records might be stored.

A related extension would be to permit access to records by general predicates on their values rather than by a unique record ID. This would permit the algorithm to be used in a relational query environment. Relational systems generally implement access aids such as indexes that can quickly locate records by their values. In order to use $D^3$ in a relational environment, methods would be needed to maintain indexes on records that are continuously migrating among a set of distributed nodes.

# 12 Related Work

Replication is a well-known technique for increasing the availability of data stored in a database. Traiger *et al* [Trai 82] introduced a model for transaction semantics in distributed database systems, and Bernstein and Goodman [Bern 83] presented a theory for proving the correctness of replication algorithms.

To provide high availability, some systems rely on special hardware such as shared memory or dual-ported disks. Tandem's "Non-Stop"$^{(TM)}$ system [Borr 81], for example, guards against loss of data by means of a "mirrored disk" approach. Another example is the Redundant Array of Inexpensive Disks (RAID) approach [Patt 88], which "stripes" each data block across sectors on multiple disks, and writes an additional "parity sector" on another disk, thus enabling reconstruction of data in the event of a single disk failure.

Other systems, for example Teradata [Tera 85], replicate data in a distributed system without special hardware. A large number of distributed data replication algorithms have been published [Bhide 90, ElAb 86, Giff 79, Hsiao 90]. While some of these algorithms address the problem of partitioning the data in a way that avoids an unbalanced workload after a failure, they all rely on a static mapping of the replicas of an object or data block to the nodes in the system. Thus, none of the above algorithms deals with the problem of dynamically redistributing data.

The ability to redistribute data without interruption of service is the central feature of the $D^3$ algorithm presented here. It allows automatic migration of data for load-balancing purposes, and enables the system to be reconfigured by adding or deleting nodes without interruption of service. Furthermore, after a failure the algorithm allows the system to recover to a "safe" state in which it is protected against additional failures without waiting for recovery of the failed node.

# References


[Bern 83] P.A. Bernstein and N. Goodman. "The Failure and Recovery Problem for Replicated Databases". *Proc. of the 2nd Annual ACM Symposium on Principles of Distributed Computing,* Aug. 1983.

[Bhide 90] A. Bhide, H. Hsiao, A. Jhingran, and A. Goyal. "Asynchronous Replica Management for Shared Nothing Architectures". IBM Research Report RC 16403, T.J. Watson Research Center, Yorktown Heights, NY, Dec. 1990.

[Borr 81] A. Borr. "Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing". *Proc. International Conference on Very Large Data Bases*, 1981.

[Cham 92] D.D. Chamberlin and F.B. Schmuck. "Dynamic Data Distribution in a Shared-Nothing Multiprocessor Data Store". IBM Research Report RJ8730, Almaden Research Center, April 1992.

[ElAb 86] Amr El Abbadi, Dale Skeen, and F. Cristian. "An Efficient, Fault-Tolerant Protocol for Replicated Data Management". IBM Research Report RJ 4851, Almaden Research Center, June 1986.

[Giff 79] D. K. Gifford. "Weighted Voting for Replicated Data". *Proc. 7th ACM-SIGOPS Symp. on Operating System Principles*, Dec. 1979, pp. 150-159. pp. 223-242.

[Gray 79] J. Gray. "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979.

[Gray 81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. "The Recovery Manager of the System R Database Manager". *ACM Computing Surveys*, Vol. 12, No. 2, June 1981, pp. 223-242.

[Hsiao 90] H. Hsiao and D.J. DeWitt. "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines". *Proc. of the 6th International Conf. on Data Engineering*, Los Angeles, Feb. 1990.

[Mohan 89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". IBM Research Report RJ 6649, Almaden Research Center, January 1989.

[Patt 88] D. Patterson, G. Gibson, and R. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". *Proc. of ACM-SIGMOD Conference*, Chicago, May 1988.

[Tera 85] Teradata Corp. "DBC/1012 Database Computer System Manual, Release 2.0". Document No. C10-0001-02, Nov. 1985.

[Trai 82] I. Traiger, J. Gray, C. Galtieri, and B. Lindsay. "Transactions and Consistency in Distributed Database Systems". *ACM Transactions on Database Systems*, Vol. 7, No. 3, Sept. 1982, pp. 323-342.

## Summary of Messages and Operator Commands

Primary messages received by a T-node:

```
Begin transaction ()
     returns (returnCode, TranID)
Get (TranID, RecID) returns
     (returnCode, data)
Store (TranID, RecID, Data)
     returns (returnCode)
Delete (TranID, RecID)
     returns (returnCode)
Commit (TranID)
     returns (returnCode)
Rollback (TranID)
     returns (returnCode)
```

Secondary messages received by any node from a T-node:

```
Simple Get (TranID, RecID)
     returns (returnCode, data)
Simple Update (TranID, RecID, Data)
     returns (returnCode)
Simple Create (TranID, RecID, Data)
     returns (returnCode)
Simple Delete (TranID, RecID)
     returns (returnCode)
Prepare to Commit Local Transaction (TranID)
     returns (returnCode)
Commit Local Transaction (TranID)
     returns (returnCode)
Rollback Local Transaction (TranID)
     returns (returnCode)
```

Messages received by any node from the Coordinator:

```
Delete old PF (PF-id)
     returns (returnCode)
Prepare for new PF (newPF)
     returns (returnCode, transStatusList)
Activate New PF (PF-id, transStatusList)
     returns (returnCode)
Shutdown ()
     returns (returnCode)
Continue (transStatusList)
     returns (returnCode)
Report Status () returns
     (returnCode, PF-list, transStatusList)
Active PF's (PF-list)
     returns (returnCode)
Report Load Statistics()
     returns (returnCode, busyMeasure)
```

Messages received by the Coordinator from any node:

```
Inactive PF (PF-id, NodeID)
     returns (returnCode)
Failed Node (NodeID)
     returns (returnCode)
```

Messages received by any node from any node:

```
Migrate Block (BlockID, Data)
     returns (returnCode)
Lock Released (LockID)
     returns (returnCode)
You're the new Coordinator (OldCoord)
     returns (returnCode)
Query PF ()
     returns (currentPF)
Flush Transaction IDs (TranIDList)
     returns (returnCode)
```

Operator commands received by the Coordinator:

```
Add Node (NodeID)
Delete Node (NodeID)
Resume ()
```

Operator commands received by any node:

```
Initialize (NodeID, list of other NodeIDs)
Reset ()
```