CUCS-236-86

Replication and Nested Transactions in the Eden Distributed System

Calton Pu

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Hardware redundancy in distributed systems offers the potential for increased availability and performance, but this requires software support if the full potential is to be realized. We have designed and implemented two mechanisms for such support. The first provides crash-resistant resources, replicated transparently and consistently to increase the availability of distributed data. To update multiple copies despite down nodes, we have introduced the Regeneration method, used in the implementation of a replicated system directory. Regeneration restores inaccessible copies elsewhere in the network, maintains the availability of resources, and adapts to configuration changes.

The second mechanism is a system supporting nested transactions, which can manage the complex failure modes in a distributed system, synchronize concurrent resource access internal to applications, and facilitate safe module composition. In the tree-structured nesting, each transaction has a Transaction Manager (TM), responsible for the concurrency control and crash recovery of its subtransactions. Many concurrency control and recovery techniques can be combined in this TM Tree design framework. We chose locking and versions for the first implementation. Using Eden objects and the replicated directory, our nested transactions provide consistent concurrent access to location-independent, crash-resistant resources.

In summary, the principal contributions of this research are the Regeneration method and the TM Tree framework. Regeneration uses the separation of hardware repair from data restoration to increase replicated data availability. TM Tree composes existing techniques to derive many different designs for nested transactions. Both have been proven in the design and implementation of actual systems.

"A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, University of Washington.

"This work was supported in part by the National Science Foundation under grant No. MCS-8004111.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Distributed systems may offer increased availability with independent failure modes and improved performance with concurrent execution. However, distributed applications with high concurrency and availability need more than just hardware redundancy. Distributed data must be kept consistent and available despite partial failures in the network. The probability of all data resources distributed over N nodes in a network being accessible is the product $\prod_{i=1}^{N} P(i)$, where $P(i)$ is the probability of node i being accessible. Without data redundancy, partial failures decrease the probability of successful executions of distributed applications.

We have introduced the Regeneration method for data replication. In some cases, data remain inaccessible for relatively long times – for example, in hard failures requiring manual repair. Regeneration separates the repair of disabled hardware from the restoration of data. Through a probabilistic analysis, we show that Regeneration takes advantage of this separation to provide more availability than other data replication techniques, which wait for hardware repair. Using Regeneration, we have implemented a replicated directory system, which is used in the replication support integrated into a nested transaction mechanism.

Nested transactions organize operations on distributed data into atomic sets, which are useful in several ways. First, partial failures in such an atomic set revert distributed data to their original state, undoing intermediate changes. This simplification facilitates the handling of the 2^N possible failure combinations in a network of N nodes. Second, concurrent resource access from different atomic sets are synchronized automatically, promoting increased concurrency in distributed applications. Third, these atomic sets of operations do not interfere with each other, allowing safe and reliable combination of distributed software.

To support nested transactions, we have designed and implemented the Eden Resource Management System (ERMS). Unlike previous proposals, which extend specific concurrency control and crash recovery methods, we shall here separate the concerns of nesting from particular implementation techniques. Normal, single-level concurrency control and crash recovery techniques are implemented at each level; careful composition of these modules provides resource access synchronization, failure isolation, and uniform syntax. In addition, the resources are location-independent Eden objects, which may optionally be replicated.

The remaining sections of this chapter introduce our model of distributed systems and the

Eden testbed used in the implementation. In chapter 2, we introduce the transaction concept and its generalization to nested transactions; then, we summarize techniques for the implementation of transaction systems. In chapter 3, we divide the data replication problem into three parts: multiple-copy updates, network partitions, and transparent access. Using this analysis, known replication techniques are described and compared. Chapter 4 introduces the Regeneration method for consistent multiple-copy updates. Chapter 5 describes our design and implementation of a replicated directory system using Regeneration. Chapter 6 describes our design choices and implementation of nested transaction support. Chapter 7 generalizes ERMS to a general design framework for nested transactions. Finally, chapter 8 concludes the thesis with summaries of contributions and future work.

1.2 Model and Definitions

Our model of a distributed system is a network of computers with independent failure modes. In our algorithms, design, and implementation, we give no consideration to malicious behavior, such as Byzantine faults [71]. Our computers are either *up* and running, or *down* and inaccessible. However, network links may fail, causing network partitions. Within a partition, all resources on up nodes are accessible. We define *availability* as the probability of a resource being accessible at any given time. The number and identification of nodes running and in communication with each other constitute the *system configuration*. We assume that approximate system configuration data may be obtained.

Resources are encapsulated in objects, which can support arbitrary operations. Although the model imposes no restrictions on the type of resources, in this dissertation we focus on resources containing data. To simplify the implementation and presentation, we divide the resource access operations into two classes. *Writes* modify the resource's state. *Reads* get values that are a function of the resource's state. Usually, our resources are data rather than hardware.

Our model of a database is a set of resources residing in stable storage. The objects may be bound by some rules, called *consistency constraints*. If objects all follow the consistency constraints, the database is in a consistent state. A *transaction* is a collection of operations on resources to be performed atomically. We assume the transactions take the database from one consistent state to another consistent state.

This model adequately describes many practical distributed systems, including those discussed in chapters 3, 6, and 7. Therefore, the analysis of replication in chapter 3, the Regeneration method in chapter 4, and the discussion of nesting by composition in chapter 7 are all applicable to a wide range of distributed systems, transcending the applications implemented in the Eden system (section 1.3).

1.3 The Eden System

Both replication support and nested transactions mechanism were implemented on top of Eden [2,17,53], an experimental distributed operating system used as a test bed. Eden encapsulates resources in objects and provides location-transparent invocations between objects.

Eden provides a unique combination of features that are extensions of those found in other systems. Eden supports user-defined and extensible objects, which communicate with each other via invocations, in the form of Remote Procedure Calls [13]. In comparison, the Apollo Domain system [54] is object-oriented, but its objects are pre-defined; Hydra objects [86] are defined by the users, but Hydra runs on a centralized multicomputer system. Eden objects can also migrate from one node to another; for example, computation may be distributed for load balancing, or replicas of a resource placed on separate machines.

Eden objects contain data, procedures, and active processes. They encapsulate files, programs, and other resources. The designers of an Eden object can define arbitrary operations to which the objects respond. Each object is invoked through a capability, which is a system-wide unique identifier together with a set of access rights defined on the invocations it supports. Remote invocations have the same syntax and semantics as local ones. Objects may have an active form in main memory and a passive representation on disk. An active form of the object services an invocation immediately. If an invoked object has only a passive representation, the Eden kernel activates the object. Each Eden object is an instance of an *Edentype*, a program written in the Eden Programming Language [16], which is an extension of Concurrent Euclid [43].

Eden objects are the building blocks with which we have implemented the replicated directory (chapter 5) and the nested transaction mechanism (chapter 6).

Chapter 2

Transactions for Consistency

2.1 The Transaction Concept

The transaction concept [39] underlies most of the work described in this dissertation. In this section, we define the concept and extend it to cover nested transactions [62]. Transaction implementation techniques are described in section 2.2 (concurrency control) and section 2.3 (crash recovery). These techniques will be used in our implementation (chapters 5 and 6), and discussed in hypothetical designs (chapter 7).

2.1.1 Single-Level Transactions

A *transaction* is a collection of operations on resources that is "all or nothing"; either all or none of the operations result in permanent resource changes. Transactions that do not alter the data in resources are called queries, read-only transactions, or read transactions. Transactions that also write to resources are called updates or write transactions. We assume that an update takes the database from one consistent state to another consistent state.

The "all or nothing" property, also called atomicity, implies that transactions executing concurrently do not see intermediate results produced by other transactions. Techniques called *concurrency control* methods, described in section 2.2, synchronize interleaved transactions so they observe the concurrency atomicity requirement. The main benefit of concurrency atomicity is that transactions may execute in parallel, and as long as the database starts from a consistent state, all transactions are guaranteed to read consistent values from the database. The intermediate results, which may be temporarily inconsistent, are hidden by the concurrency control.

Atomicity also requires that transactions interrupted by machine crashes should not leave any intermediate, potentially inconsistent results. *Crash recovery* techniques, described in section 2.3, achieve this goal, called reliability atomicity, by either rolling the transactions back to their initial state, or forward to their final state. The combination of concurrency atomicity and reliability atomicity maintain the database consistency despite concurrent accesses and machine failures.

Unlike traditional definitions where a transaction is a sequence of operations, we allow concurrency within a transaction. Multiple threads of control, called *processes*, may execute the operations in parallel to take advantage of a distributed system. Although atomicity guarantees consistent access and crash recovery between transactions, more structure is needed within

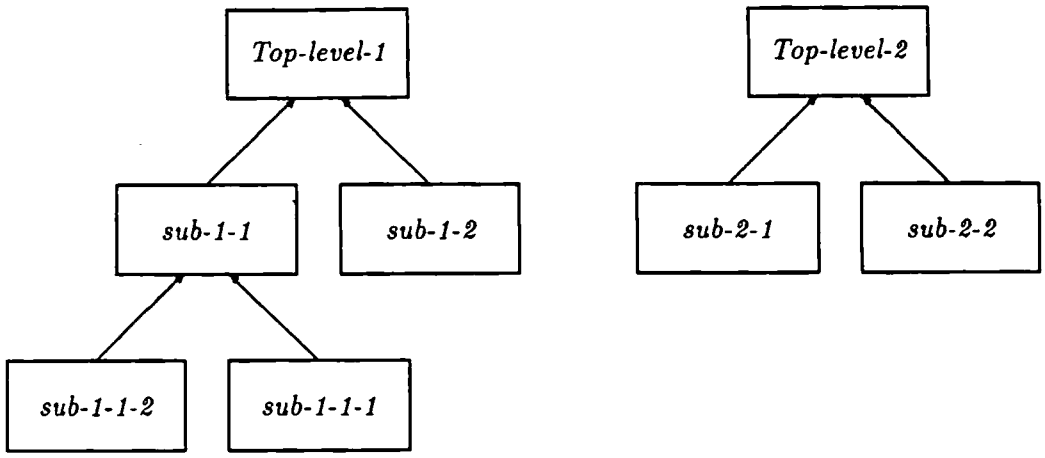

Figure 2.1: Standard Model of Nested Transactions

a transaction. To synchronize resource access by internal processes and recover from individual process failures, nested transactions introduce a natural extension of structured atomicity into transactions.

2.1.2 Nested Transactions

Subtransactions are subcollections of operations performed atomically within a transaction. Eswaran et al. [30] defined a transaction as a sequence of operations. Unlike their transactions, our subtransactions may run concurrently, as in the standard nested transaction model defined by Moss [61]. The top-level transaction and its subtransactions form a tree (figure 2.1). The top-level transaction is at the root of the tree; subtransactions are the internal nodes and leaves. We will use tree terminology to denote the relationship among transactions: ancestors and descendants, parent, siblings, and children. Intermediate states produced by a subtransaction are hidden from its siblings and parent. If a subtransaction aborts, none of its operations leave any effect, but the parent and siblings may continue executing and commit eventually.

In distributed systems, there are several advantages in using the structure of nested transactions. First, subtransactions abort independently of their parent and siblings, helping the programmer to recover from the 2^N possible failure combinations in N nodes. Second, resource accesses from different processes are automatically synchronized for concurrent subtransactions. Third, composition of atomic subtransactions into a structured enclosing transaction facilitates construction and maintenance.

In the rest of this chapter, we describe the implementation techniques of single-level transactions. Since our nested transaction design and implementation rely only on the techniques summarized in the following sections (2.2, 2.3), the discussion of other early designs of nested transactions will be summarized later in section 6.7.

	read	write
read	Y	N
write	N	N

Table 2.1: A Simple Lock Compatibility Table

2.2 Concurrency Control

The commonly accepted correctness criterion for concurrent database access is *serializability*, defined as an execution of concurrent transactions that produced the result equivalent to a sequential execution of those transactions [68]. A concurrency control method guarantees the serializability of resource access despite concurrent access from several transactions.

In this section, we provide background material on concurrency control for our work on both replication and nested transactions. We describe *two-phase locking* in section 2.2.1, *timestamps* in section 2.2.2, and *optimistic concurrency control* in section 2.2.3. Readers familiar with these concurrency control methods can safely skip to section 2.3. The survey of these well-known concurrency control methods is based on works by Kohler [48], Bernstein and Goodman [9], and Mohan [60]. Other concurrency control methods found in the literature, such as tickets, conflict analyses, and reservations, have been omitted since they are not pertinent to this dissertation.

2.2.1 Two-Phase Locking

One way to synchronize concurrent access to a resource is to lock the resource to prevent conflicting access. Before a transaction can access a resource, it requests a lock on the resource from the system lock manager. If the lock is granted, the lock manager guarantees that the transaction has a consistent view of the resource. When the transaction ends, the locks it held are released.

The simplest way to achieve consistency is to allow only one transaction to access the resource at any given time. The exclusive access, sufficient for update consistency, is not necessary when transactions are only reading the resource. A common lock convention allows either multiple, concurrent readers (shared lock) or an exclusive writer (exclusive lock). Readers do not alter the resource state, so they are able to share access without compromising consistency. In contrast, a writer may take a resource through some intermediate, inconsistent states, therefore the resource must be exclusively locked. Table 2.1 shows the compatibility between the two kinds of locks. A write lock is granted only if no other transaction holds locks of any kind on the resource. A read lock is granted whenever the resource is not exclusively locked.

Although these rules guarantee the consistency of each resource, more care is needed when a transaction accesses multiple resources. Eswaran et al. [30] have proved that all transactions must be two-phase in order to assure multiple resource access consistency. A two-phase transaction locks all the resources it needs before unlocking any of them. In other words, the transaction can be divided into two phases, the growing phase, in which it locks all its resources, and the shrinking

phase, when it only unlocks. Such two-phase transactions are serializable, so two-phase locking maintains database consistency.

Avoiding the formal treatment, which can be found in the literature [30,68], we use an example to illustrate the way two-phase locking works. Let transactions T_1 and T_2 be two-phase according to the above description. If T_1 holds a lock on a resource, and T_2 requests an incompatible lock on the same resource, two actions are possible. We can make T_2 wait, or we can reject its request. We will consider waiting in this section, and request rejection in section 2.2.2. If T_1 finishes and releases its locks, T_2 gets the desired lock, and everything ends well. However, in an alternative scenario, T_1 locks resource A exclusively at the same time T_2 locks resource B; and later T_1 tries to lock B, while T_2 attempts to lock A. Transactions T_1 and T_2 will keep waiting for each other, in a situation called *deadlock*.

There are several ways to handle deadlocks. First, deadlocks may be avoided by preclaiming all resources at the very beginning of every transaction. The disadvantage is that some resources may be locked longer than necessary, since the transaction may access only a subset of all resource in a given run. Its advantage is that since all resources have been locked, the transaction will not be aborted because of resource contention. So by preclaiming all resources, we trade system concurrency for freedom from deadlocks.

Second, deadlocks may be detected by finding cycles in the transaction wait-for graph, defined as follows:

- Each vertex in the wait-for graph corresponds to a transaction.
- Each arc in the wait-for graph links a transaction with a blocked request for a resource to the transaction currently holding the lock on the resource.

The lock manager maintains the wait-for graph according to the lock requests. If a cycle is formed in the wait-for graph, the transactions in the cycle are waiting for each other indefinitely. To break the deadlock, the lock manager uses some kind of heuristics to decide which transaction in the cycle must be aborted. In the next section, we outline a third deadlock resolution technique where unsuccessful lock requests are rejected instead of blocked. Many papers [60] exist on deadlock detection and resolution, and we will use these known techniques in our work.

Thus far, we have described two-phase locking of centralized databases. The distribution of lock management and deadlock detection in a distributed database requires further development. Since distribution in Eden is hidden by location-independent capabilities, we omit the discussion on these topics. For more details, we refer interested readers to the referenced survey papers [9,48,60].

Additional rules on top of two-phase locking have been proposed by Moss [61,62], and implemented in Argus [56] and LOCUS [63,84], to control concurrency in nested transactions. These extensions will be discussed in section 6.7.

2.2.2 Timestamps

In contrast to two-phase locking where each resource is locked to prevent conflicting access, the timestamp method serializes the transactions directly. Each transaction is given a system-unique number, called a timestamp, which determines its place in the serialization. From the timestamp

attached to each read and write request, resource access conflicts are detected and resolved as follows. Instead of a lock, each resource remembers the timestamps of the latest reader (TS_r) and writer (TS_w). If a read request has a timestamp earlier than the resource's TS_w, it means that the writer with timestamp TS_w, which must follow the reader, has overwritten the value the reader needed. Since the reader has been superseded, it must be aborted. Similarly, a write request fails if its timestamp is earlier than either TS_w or TS_r.

In a distributed database, several techniques to generate global timestamps are known. The existence of a global clock certainly solves the problem. Lamport [50] has shown that a central clock is not required, and some communication between nodes can maintain global time. In cases where strict global time ordering is not necessary, system-unique identifiers can be more easily produced by concatenating local timestamps with node identification numbers.

The usual way to implement timestamp concurrency control relies on versions. Each resource is made of a series of versions, created by successive update transactions writing on the resource. At each moment in time, the resource is represented by the most recent version at that time. For example, let us consider a resource with three versions, V_1 created at T_1, V_2 created at T_2, and V_3 created at T_3, where T_i precedes T_j, $i < j$. From T_1 to T_2, reading the resource would get V_1, and from T_2 to T_3, V_2 represents the resource. The resource becomes V_3 after T_3. Since read requests with earlier timestamps can read earlier versions instead of aborting, the versions decrease the number of aborts. Versions are also useful for crash recovery and will be discussed further in section 2.3.1.

The main technical difference between two-phase locking and timestamps is that the serialization order determined by timestamps eliminates deadlocks. Actually, timestamps can be used for deadlock resolution. Rosenkrantz et al. [75] have proposed the wait-die and wound-wait deadlock resolution schemes based on timestamps. In systems using wait-die or wound-wait, unsuccessful lock requests do not block, but return with the timestamp of the conflicting lock holder. Based on the comparison between the returned timestamp and its own, the transaction decides whether to continue or abort.

The basic timestamp method described above assigns the timestamps at the beginning of each transaction. Two transactions, $\mathbf{T_1}$ and $\mathbf{T_2}$, may be serializable in a particular order, for instance, $\mathbf{T_1}$ followed by $\mathbf{T_2}$. If $\mathbf{T_1}$ happens to have been assigned a timestamp later than $\mathbf{T_2}$, one of them would have been aborted. To remedy these situations, Bayer et al. [7] introduced dynamic allocation of timestamps, or timestamp intervals. The timestamp interval method does not assign timestamps a priori; it allows transactions to run optimistically, with a range of possible timestamps in an interval. When conflicts arise, the timestamp intervals of the conflicting transactions are refined to make the serialization explicit. In the above example, this improvement allows both $\mathbf{T_1}$ and $\mathbf{T_2}$ to commit.

2.2.3 Optimistic Concurrency Control

Optimistic concurrency control methods [21,49] assume that the probability of conflicts between transactions is low. Consequently, greater concurrency may be obtained by postponing the conflict detection and possible abort of transactions to the last possible moment. Basically, they analyze the transaction dependency graphs just before each transaction commits. The transaction depen-

dency graph is defined by the following rules to show the relationships between the transactions running in the system.

- The resources read by a transaction is called its input, and the resources being written, output.
- If a transaction's input contains a resource from another's output, they are linked by an arc in the transaction dependency graph.

An acyclic graph shows a partial ordering of transactions from which an equivalent serial schedule can be constructed. A cycle in the graph indicates conflicting transactions that cannot be serialized. The optimistic concurrency control method detects the formation of cycles in the graph, and prevents the transaction closing the cycle from committing, thus maintaining the graph of committed transactions acyclic. An acyclic graph induces a total ordering of the nodes, so transactions on an acyclic dependency graph are serializable.

The optimistic method differs from locking in both amount and moment of aborts. Compared to locking, the optimistic method allows more concurrency, since no restrictions are placed on the transactions. However, the price paid for the added concurrency is that aborts now waste whole transactions instead of just parts of them. In contrast, two-phase locking may abort a transaction during the growing phase, but once into the shrinking phase, no aborts due to conflicts are possible.

In contrast to the optimistic method, which aborts a transaction only when absolutely necessary, some methods using timestamps tend to abort more transactions. However, it is not obvious how the optimistic method compares to a more refined timestamp method such as timestamp intervals. Other ongoing work [66] is studying the problem.

2.3 Crash Recovery

In this section, we provide background material on crash recovery, which guarantees transaction atomicity despite node crashes. Readers familiar with the literature can safely skip to chapter 3. The survey of these crash recovery techniques is based on work by Bernstein et al. [11], Haerder [40], Kohler [48], and Verhofstadt [81].

The basic idea underlying most recovery methods is to save the database state on stable storage before the transaction changes it. Since the database state is consistent at the beginning and the end of each transaction, these are the best occasions to save state information. First, before a transaction starts, the database state exists on disk, so if the transaction aborts, the database can be rolled back to its initial state. Second, before the transaction commits, the database state produced by the transaction is saved on disk, so if the node crashes and the database is lost, the committed state can be recreated by rolling forward from some earlier checkpoint.

Two basic techniques to implement the abstract algorithm above are summarized here. The first stores multiple *versions* of each resource at critical moments, so recovery consists simply of choosing the right version. The second writes the recovery information on a sequential *log*, requiring more sophisticated recovery.

2.3.1 Versions

The main idea of version-based recovery is to maintain a consistent picture of the database at all times. The temporary inconsistent states created by update transactions are hidden in new versions of each resource [48,81]. Consequently, the database always has the most recent version of all resources written by the committed transactions. Aborting a transaction leaves the previous version in place, and committing a transaction switches from the old version to the new one. Care must be taken to ensure atomic switch of all resources written by a transaction, even if the machine crashes in the middle of such operation.

One way to achieve atomicity of the switching relies on idempotent operations [51]. An idempotent operation can be executed many times, but its effects happen only once, during the first successful execution. When re-executed, it has no effect, and returns the results obtained by the first execution.

Atomic commit with idempotent switch operations is relatively simple. At the beginning of transaction commit, we write a commit record atomically to disk. The commit record contains all the switch operations which must be executed to make the new versions official. If the commit is interrupted by a machine crash, a recovery procedure is executed before the transaction processing resumes. The recovery simply restarts the idempotent switch operations from the beginning. Once all new versions have been installed, the commit completes. This is the method chosen for the atomic commit of our nested transaction support (chapter 6).

An alternative way is to build a hierarchical structure pointing to the resource [64]. A tree-structured directory stores the pointers to the versions. When the transaction commits, a commit record containing the pointers to the new version of several resources is written. The commit protocol then proceeds to construct a new tree with the new pointers. Writing the root page of the tree is atomic and it switches to the new tree with new versions. If the commit protocol is interrupted by a crash, the recovery procedure rebuilds the new tree and completes the switch.

2.3.2 Logging

The main motivation for log-based recovery is the trade-off between normal processing overhead and recovery time. Usually, the more time we spend during the normal processing storing information for crash recovery, the less time recovery takes. If crashes are relatively rare events, we should optimize the normal processing, even though the recovery time may be lengthened somewhat. For update operations, disk accesses constitute a significant part of the cost in database operations, often the limiting factor [38]. Since writing the recovery operations to a sequential output is usually faster than to random places on a disk, logging has become the most popular method for practical databases.

The main advantage of logging is that it allows the update transactions to write in-place on the database. If a transaction does not abort, writing in-place bypasses the creation of new versions altogether. However, writing in-place introduces intermediate and inconsistent information into the database. Logging relies on a recovery manager to reconstruct a consistent database after a crash has occurred, using the information on the log. The recovery manager reads the log and repairs the database as soon as inconsistencies are detected. The normal transaction processing can resume only after the recovery manager has finished the repair.

Typically, there are two ways to save information on a log [77]. The first method writes either the old or the new value of a resource on the log in an update. In case of a crash, aborted transactions have their old values rewritten into the database, in an operation called *undo*. Committed transactions which did not propagate their results to the actual database have their new values written on the database, in an operation called *redo*.

The second method records the operation performed on the resource, instead of the values being changed. Since either the old value or the new value of the resource always exists on disk, recording the operation is sufficient for recovery. For undoing an update, a reverse operation is required to recover the old value from the current state in the database. For redoing an operation, if its result was lost in the system buffer and the database contains old values, it suffices to execute the logged operation again.

In either case, the transactions must obey the "write-ahead" protocol. Before the write in-place is executed, the transaction must ensure that recovery information needed to that write has been written on the log. If the recovery information is not in the log before the inconsistent data are introduced to the database, a crash at precisely that moment may preserve the inconsistency. Since the log record on that change has been lost, the recovery manager will be unable to repair the inconsistent database. The write-ahead protocol avoids these problems by keeping enough information to reconstruct a consistent database on log.

In addition to the write-ahead protocol, if all the write requests of a transaction are written on disk before the commit record is written, some simplifications are possible [11]. In this case, once the commit record is written, all the updates are in place. So in case of machine crashes, only undo operations are necessary to roll back aborted transactions. Similarly, if the update operations are carried out only after the commit record is written, aborting them requires no work. We only need to redo the committed transactions interrupted in their writing. If transactions are allowed to write to disk at any time, the recovery manager must be able both to undo and redo transactions from the log.

2.3.3 Distributed Commit

Thus far, we have described crash recovery in centralized databases. The distribution of logs and versions requires further development. Since distribution in Eden is hidden by location-independent capabilities, we omit the discussion on these topics. However, in a distributed transaction, its components must agree on the transaction's outcome. One such protocol is the two-phase commit protocol [9].

The two-phase commit protocol assumes there is a transaction coordinator, and subtransactions run as servers. During the first phase, the coordinator sends out messages asking the final results of the servers. Servers respond with their outcome, whether they have committed or aborted. If some servers have aborted, the distributed transaction aborts and the coordinator sends the abort message to the servers. If all servers agree on commit, the protocol enters the second phase. The coordinator writes the commit decision to stable storage and sends the commit message to all servers. The most important feature of the protocol is that all participants agree on the outcome of the distributed transaction.

Chapter 3

Replication for Availability

3.1 Crash-Resistant Resources

Definition 1 *A resource is k-crash-resistant if and only if the following three conditions hold:*

1. Accessibility: the resource remains accessible despite k nodes being shutdown.
   - Since k copies may be down, a k-crash-resistant resource must have at least $k+1$ copies distributed over distinct nodes.
2. Consistency: the resource access has one-copy semantics.
   - Writing to the resource is atomic; reading from the resource returns the value either before or after the write.
   - Between writes, a read always gets the same value.
3. Transparency: the resource access has one-copy syntax.
   - Resource access syntax is the same regardless of k.
   - Resource access syntax is independent of node crashes.

Notation: A crash-resistant resource is k-crash-resistant for some $k, k > 0$.

A 0-crash-resistant resource is non-replicated and therefore vulnerable to the crash of a single node. For crash-resistant resources, the Accessibility condition implies replication. The Consistency condition, equivalent to 1-serializability [10], stipulates that the replicated resource should contain only one value at any given moment, just like a single-copy resource. The Transparency condition means that the number and location of copies should be transparent to clients.

In published works on replication [10], the Accessibility and Consistency conditions appear routinely, but not Transparency. We impose Transparency for two reasons. The first is encapsulation. We want to hide the details of replication from clients, so clients do not have to re-implement the replication algorithms each time. The second is flexibility. Distributed systems have many independent parts, so we want a replicated resource to be adaptable to configuration changes without affecting the clients.

With crash-resistant resources, we want to increase resource availability, maintaining one-copy resource access syntax and semantics. We divide the support for crash-resistant resources into three parts: consistent multiple-copy update, network partitions, and transparent access.

The three parts of the replication problem handle different failure conditions and consistency requirements. The first part, multiple-copy update, handles consistent concurrent updates of multiple copies, recovery from nodes crashes, and availability despite node shutdowns. The second part, network partitions, deals with communication failures resulting in machines running in separate partitions, and unable to communicate. The third part, transparent access, concerns a crash-resistant root directory, which provides the mapping of resource names into their copies. We analyze the consistent multiple-copy update problem in section 3.2, briefly consider the network partitions problem in section 3.3, and discuss the crash-resistant root directory in section 3.4.

3.2 Multiple-Copy Update

Consistent multiple-copy update can be divided into two sub-problems: atomic update of replicas, and node shutdowns. To guarantee that all replicas contain the same value (one-copy semantics), we need to make each update atomic, so all copies will reflect either the old or the new value uniformly. Traditional transaction techniques are well-suited for the atomic update sub-problem, as Gifford has pointed out in his thesis. More concretely, we can use concurrency control (section 2.2) to serialize concurrent updates, and atomic commit (section 2.3) to recover from node crashes.

Insisting that all copies be updated atomically does maintain resource consistency, but this policy may prevent an update from completing because a copy is inaccessible in a down node. Additional refinements must be introduced to allow updates to proceed despite some nodes being down, making some copies inaccessible. In this section, we describe some known techniques which allow consistent and atomic update of multiple copies to complete, despite a limited number of inaccessible copies. These techniques are classified into three groups, Asymmetric, Voting, and Reconfiguration. For later comparison with the Regeneration method in chapter 4, we describe representative examples from each group, without attempting a comprehensive survey of replication techniques.

3.2.1 Asymmetric — Primary Copy

Asymmetric replication methods distinguish some copies from others, giving special functions to the distinguished ones. The Primary Copy method for data replication [3] is such an example, where one copy, the primary, is distinguished from the secondary copies. Clients read from any copy. All writes are sent to the primary, which propagates the updates to the secondary copies. Inaccessible secondary copies are removed from the chain of communication and re-initialized when they join the network.

Primary Copy is an asymmetric method for multiple-copy update, because the primary copy has a special meaning not shared by the secondary copies. If a secondary copy happens to be inaccessible, the update propagation simply bypasses it. Consequently, Primary Copy is tolerant to any combination of failures of secondary copies.

The main difficulty of the Primary Copy method arises when the primary copy becomes inaccessible. In case the primary site is down, a reassignment is in order. However, if the network has partitioned, a reassignment would compromise consistency. Minoura's True-Copy Token scheme [59] improves the primary copy method in several ways. However, the reassignment of primary copy is possible only when all involved sites have come back online. Alsberg and Day [3] have left this problem to the application programmer. In this dissertation, we will consider the Primary Copy method with a fixed primary site, because no general solution exists to create a new primary when the original is down.

3.2.2 Voting — Majority

Voting algorithms [34,80] are symmetric, in the sense that all copies are equal. The basic idea of Majority Voting [80] is to read and write subsets of the total number of copies (N). Consequently, read and write can proceed even though some copies are inaccessible. In the simplest case, to access a replicated resource, a read quorum ($RQ < N$) and a write quorum ($WQ < N$) are defined such that $RQ + WQ > N$. A read operation requires the access of RQ copies, a write operation of WQ copies. Because there must be an overlap between RQ and WQ, the most recent version will always be found and accessed. The number of inaccessible copies tolerated in a read is $N - RQ$, and in a write, $N - WQ$. If the resource needs to be read and written, the fault tolerance is $min(N - RQ, N - WQ)$.

In Voting, resource access requires a subset of copies. The advantage of accessing a large number of copies is the tolerance to Byzantine failures [71]. If the subset is large compared to N, not only the access becomes expensive, but also the fault-tolerance decreases. Since the read quorum and write quorum must intersect, they have minimal sizes bound by $RQ + WQ > N$. The trade-off is between read/write availability and cost: large read quorums allow smaller write quorums and vice-versa. An additional feature of Voting is that it guarantees resource consistency despite network partitions, topic of section 3.3.

There are several refinements of Voting methods. Gifford [36] adds weights to the votes, where each vote may weigh more or less depending on some criteria of client choice. Herlihy's General Quorum Consensus [41] increases further the number of quorum choices. Instead of the read/write dichotomy, he divides the operations on objects into three kinds: read-only, write-only, and read-write. His generalized quorums provide individual quorum choices for each kind of operation (table 3.1), allowing for more diversified availability trade-offs. If an operation is read-only, then an Initial Quorum must be gathered. Similarly, only a Final Quorum is necessary for the write-only operations. For read-write operations, the Initial Quorum is necessary for read, and the Final Quorum for write.

Gifford used Voting in the Violet Calendar System [35]. Weighted Voting was also used in an experimental study of replicated directories at CMU [19,24].

3.2.3 Reconfiguration — Available Copies

Reconfiguration methods do not rely on access to subsets to provide fault-tolerance. They change the configuration (the number and/or location) of the copies to adapt to new situations created by node crashes or recoveries. The Available Copies method [8] is an example of symmetric method

	Initial Quorum	*Final Quorum*
read-only	RQ	0
write-only	0	WQ
read-write	RQ	WQ

Table 3.1: General Quorum Consensus

based only on reconfiguration. It reads any copy and writes all accessible copies (if at least one is accessible), those being defined as copies residing on accessible nodes. When a node crashes, all copies of all data items in that node are *excluded* by updating a replicated directory containing the number and location of copies. A crashed node recovers by copying up-to-date data items and updating the replicated directory to *include* those copies on the recovering node. A copy of the directory itself may become inaccessible when a node goes down. A special *directory-include* operation updates the local copy of the directory during node recovery.

Because a resource can be read or written with only one copy, network partitions may allow different copies to diverge. This problem will be discussed further in section 3.3. Also, since only one copy is necessary for reads, the Available Copies method is vulnerable to Byzantine faults. Another important feature of Available Copies is the explicit use of a directory, facilitating the implementation of one-copy resource access syntax.

DDM [22] is a distributed database system which replicates resources using the Available Copies method. The Isis project [12] has different design and implementation strategies, but the availability provided by Isis would be the same as Available Copies.

3.2.4 Comparison With Hardware Redundancy

Redundancy has been used in hardware to increase reliability of computers for a long time [74,82]. The three main techniques are, triple modular redundancy, duplex redundancy, and stand-by redundancy. Triple modular redundancy requires three identical modules which vote for the output. Duplex redundancy uses two identical modules to detect and isolate failures. Stand-by redundancy has two kinds, the cold stand-by operates with one module, and brings up the stand-by unit when the first fails. The warm stand-by redundancy keeps all modules online, and switches off the ones that have failed.

Several of the data replication techniques we have discussed in this section have hardware counterparts. Primary copy offers only partial redundancy, since the primary copy is not replicated. The secondary copies are similar to warm stand-bys, adding read availability. Voting generalizes the triple modular redundancy. Available Copies method is the software analog of stand-by redundancy. In DDM [23], online copies are maintained up-to-date if they remain accessible, like warm stand-bys. Offline copies are brought online when necessary, like cold stand-bys.

3.3 Network Partitions

Network partitions happen when some communication links fail, while individual nodes and parts of network keep running. We consider the partitioning to be a temporary situation, and each partition an integral component of the distributed system. Consequently, we want to keep resources consistent across partition boundaries.

The main difficulty is that nodes from different partitions cannot communicate with each other. So if the update activity continues independently in each partition, copies may diverge with different update histories. In face of network partitions [28], there are two choices in handling resource consistency, pessimistic, briefly described in section 3.3.1, and optimistic, summarized in section 3.3.2.

3.3.1 Pessimistic Approach - Voting

The pessimistic approach is conservative, allowing updates in only one distinguished partition. The distinguished partition can be chosen in many ways. For example, majority voting chooses the majority partition, while Primary Copy chooses the partition in which the primary copy is located. In the distinguished partition, reads and writes proceed as usual, but in the other partitions updates are disallowed. When the partitions merge, the updates are propagated from the distinguished partition to the other partitions.

The pessimistic approach maintains update consistency since read and write transactions are still serializable. In the distinguished partition, transactions proceed as usual. In the other partitions, all read transactions are serialized before any updates which occurred after the partitioning. Consequently, copies in other partitions may become out-of-date, but updates on the "official" partition will prevail when partitions merge. The resource consistency is preserved, but we lose resource update availability in all partitions except the distinguished one. In addition, for majority Voting, if we cannot gather a majority of votes in any partition, no updates are allowed anywhere.

Dynamic Voting [25] improves on the simple (and weighted) majority voting because it allows resource update when further partition occurs. Basically, the number of majority votes is not fixed as the majority of total number of copies, but it is defined as the majority of the accessible copies. The redefinition of majority quorum follows the detection of a network partition, in the partition that maintained a majority. So if each network partition left a majority survivor, systems using Dynamic Voting may operate with as few as two nodes. Only one partition is, however, allowed to update a resource at any given moment. Also, if the network partitions into three equal portions, Dynamic Voting will not allow updates in any of them.

3.3.2 Optimistic Approach - Merging

The second approach is optimistic -allowing each partition to update the resource independently. Resource availability is increased, but, when partitions merge back into the network, ways must be found to handle conflicting updates. This problem is very difficult in general. For example, usually it is impossible to merge two different versions of a text file without loss of information.

However, for specific operation classes, such as the commutative operations, merging algorithms already exist. Promising contributions to operation classification and corresponding algorithms include dissertations by Davidson [27] at Princeton, Faissol [31] and Thiel [79] at UCLA, Wright [85] at Cornell, and many other papers [18,26,33]. Some work on the detection of inconsistencies, such as version vectors [69], also has been done.

Optimistic approaches to network partitions cannot be used in conjunction with some multiple-copy update methods, such as Primary Copy and Voting. The reason is that Primary Copy distinguishes the primary copy, and consequently also the partition containing it. Similarly, Voting distinguishes the partition with the majority of votes. However, Available Copies is a multiple-copy update method that may be combined with the optimistic partition merging algorithms. Since the optimistic algorithms allow updates to proceed in many partitions, resource availability will be higher than pessimistic approaches implied by Voting or Primary Copy.

3.4 Replica Location

Techniques to handle multiple-copy updates and network partitions enable a client to create and use a consistent, replicated resource. The third condition for a crash-resistant resource is Transparency. Without it, the use and access of replicated resources is cumbersome. For example, giving the clients the number and location of copies makes future changes difficult, either as to the number or the location of copies. In addition, each client would have to implement the algorithms to handle multiple-copy updates, and maybe network partitions.

Ideally, clients should be able to access crash-resistant resources transparently, as if they were "normal" resources. There are two advantages in the encapsulation offered by transparent access. The first is flexibility, since the number of location of copies can be changed without affecting the clients. In particular, the same syntax would be used for both replicated and non-replicated resources. The second advantage is a more general service, since any client can use the replicated resources without implementing replication algorithms themselves. In providing transparent access independent of failures and resource reconfiguration, the main mechanism is a mapping of resource names into their corresponding replicas.

3.4.1 The Mapping and Its Root

To provide Transparency on behalf of the client, a mapping must translate the resource name into its copies. The client would always use the resource name, which is the same regardless of the number and location of the copies. An intermediary mechanism (for example, the file system) uses the mapping to forward the client request to actual copies. Since the directory implementing this mapping is in the access path of the crash-resistant resources, it must be replicated for availability.

The directory may be replicated using the techniques described in the previous two sections, but finding the root directory presents a special bootstrapping problem. On the one hand, the root directory must be replicated for availability. On the other hand, there is no other directory to translate a unique root directory name into the replicated addresses. If we allow clients to read replicas of the root directory directly, then updating the root directory becomes more difficult. Since a potentially unlimited number of clients may hold the address(es) of the root directory in

order to use the mapping, the address(es) cannot be easily changed; otherwise a large number of clients would have to update their pointer(s) to the root. In our discussion, we assume that we want to avoid the thorny problem of changing a pointer in the possession of an unlimited number of clients.

There are three ways to provide a root. First, we can maintain a copy of root directory at each node, removing the need for a pointer. Second, we can use a **broadcast** operation to find the root. Third, we can fix the root directory to a pre-specified configuration.

Read-only access to the root can be simpler. Clients need only a superset of pointers to root, and a way to check whether a pointer is valid. If the superset if reasonably small, then they can simply check the pointers until one current copy of root is found. This method permits easy finding of one root for reading, but another method is needed to find all copies of root for updates.

3.4.2 Full Redundancy

One way to implement a root directory is to replicate it fully. Bernstein and Goodman [8] describe a replicated directory in which each update is applied to all accessible directory replicas. In this scheme, all update messages to the root directory are sent to all operating nodes. But the clients only read the local copy of the root directory. If a node was down when the root was updated, its copy of the root is carefully re-initialized before its node becomes accessible and clients are allowed to run on that node.

Full redundancy is expensive in terms of disk space and update traffic. Each node must store a copy of the root, and execute the updates on it. On the other hand, hierarchical directory structures may reduce the size of and the need to update the root. Since root directories are consulted often, having local access at every node may even pay off because of reduced network traffic for reads.

Fully replicated, the configuration of the root directory follows the system configuration. The clients always read the local copy, and write to all accessible replicas. During network partitions, the root directories of different partitions may diverge. Any one of the partition handling techniques delineated in section 3.3 will solve this problem.

3.4.3 Broadcast

If we choose to replicate the root only partially, some nodes will not have a copy of it. Gifford [34] has suggested *broadcast* as the mechanism to locate replicas of the root. Typically, clients need not be aware of the root configuration a priori, since they can broadcast a message asking for the root. The network must understand the broadcast message for the root directory, and forward the message to the current copies of the root. From the copies which answered the call, the client selects a current copy. Any information on the copies of the root is treated as a hint, validated by access. If the hint proves to be wrong, the broadcast locates the new copies.

Because the root is found dynamically through the broadcast, a reconfiguration merely invalidates some hints. Consequently, copies of the root may be deleted and created. Using a broadcast mechanism eliminates the need to replicate the root everywhere. Only a subset of nodes will have resident copies of the root. So trade-offs between disk space and network traffic to access the root become possible.

Replication Proposals	Gifford Thesis [34]	Bern./Good. [8] DDM [22]	LOCUS [84] root directory
Multiple-Copy Update	*Voting*	*Available Copies*	*Primary Copy*
Network Partitions	Pessimistic	- Open -	Pessimistic
Root Location	Broadcast	Full Redundancy	Full Redundancy

Table 3.2: Analyzing Concrete Proposals

3.4.4 Well-Known Fixed Configuration

Another method to implement the root directory is to place its copies on a number of well-known fixed nodes. If a broadcast mechanism is not available in the network, and disk space is scarce –precluding full redundancy–, this may be the only alternative. The main problem of fixing the root to a specific configuration is the difficulty in the reconfiguration of the root, which may restrict the utility of this method to experimental systems with a limited life expectancy.

There are no problems with reading, since the copies are hardwired to well-known nodes. However, it is harder to update the root if one of the nodes happens to be down. The simplistic solution aborts the update. A more elaborate solution ignores the down copy, and brings it up-to-date during that node's recovery, just as in the full-redundancy case. Voting is a more expensive solution to read and write the root directory. Voting provides fault-tolerance, but also makes remote access mandatory in each read.

Forward pointers [32] could be used to move the content of the root directory, but the root address remains the same. Moreover, all nodes in the forwarding chain must be up and running for the access to succeed. Therefore, the use of forwarding addresses decreases the availability of the root directory, although it allows the content of the root to be moved.

The main problem with a fixed configuration is that the root configuration cannot be changed easily. The nodes which contain the root will have to stay as long as the system itself. Substitution of new hardware is sometimes possible, but the number of copies is hard to change.

3.4.5 Summary

Compared to the many replication techniques proposed for multiple-copy updates and handling of network partitions (one-copy semantics), transparent access to crash-resistant resources (one-copy syntax) deserves more attention. Table 3.2 compares the root directory locations of some proposed or implemented systems. The Bernstein and Goodman proposal [8] explicitly excludes network partitions from their model, but it seems possible to adapt either the pessimistic or the

optimistic approach to handle partitions in the concrete system DDM [22].

Each one of the techniques to find the root has advantages and disadvantages. The broadcast mechanism, if available in the system, can be used to find the root in any replication scheme. However, for large distributed systems, the implementation of broadcast presents difficult problems. A fully redundant root directory allows read-one/write-all access. Since the cost in disk space and update traffic will escalate with the scale of the system, a hierarchical structure should be used to minimize the size and updates of the root. In this case, reading of the root becomes cheaper as the system grows larger. Finally, a well-known fixed set of copies for root can be used, but this presents serious difficulties in the reconfiguration of root. Although this limitation may be acceptable in experimental systems, the growth of a distributed system will be hampered by fixing the set of copies for root.

Chapter 4

The Regeneration Algorithm

4.1 The Regeneration Algorithm

The Regeneration algorithm is motivated by the fact that data restoration can be done independently of hardware repair. Each multiple-copy update method described in chapter 3 –Primary Copy, Voting, and Available Copies– considers copies resident on down nodes to be inaccessible until the nodes are repaired. The main idea of Regeneration is that we can replace those inaccessible copies with new replicas on running machines, even before the down nodes are repaired. In addition to maintaining replica consistency, Regeneration also increases resource availability if hardware repair takes longer than the making of a new copy.

4.1.1 Definitions and Assumptions

A *replicated resource* consists of several identical copy objects. Copies may become discrepant during an update. A *replica* is a copy reflecting the desired state of the replicated resource.

Each replicated resource has some *configuration data* stored in a directory: the number, names, and location of its replicas. A resource whose configuration data may be changed is *reconfigurable*. The directory is replicated to increase its own availability. The directory's configuration data may thus form a hierarchy with many levels of indirection and a root.

We assume that an underlying transaction mechanism, such as the ones described in chapter 2, will perform an atomic update of a set of replicas. We also assume that the directories are reconfigurable, including the root. In chapter 3, we have divided the replication problem into three parts: multiple-copy update, network partition, and root directory. The Regeneration algorithm solves the multiple-copy update problem. Network partitions and root directory will be discussed in section 4.2.

4.1.2 Conceptual Algorithm

The key idea of the Regeneration algorithm is to make new, accessible replicas to replace the inaccessible copies. Two benefits arise from regeneration: first, replacing the inaccessible copies eliminates the potential inconsistency caused by the update. Second, new replicas restore the replicated resource availability to its maximum specified level, even before the hardware is repaired.

Previous data replication methods, described in chapter 3, also maintain resource consistency, but they all wait for the hardware repair to restore their data.

Figure 4.1 describes the two basic operations: read and write of resources. These operations should be used when accessing replicated resources, analogous to the way one would read and write a non-replicated resource; each of these operations returns **abort** or **success**. There are three additional technical observations. First, application data transfer is omitted for clarity. Second, updating the directory makes it point to up-to-date replicas. Third, the directory is itself replicated using Regeneration, up to the root directory, which will be discussed in section 4.2.

4.1.3 Discussion

The Regeneration method provides high resource availability in three ways. First, reading requires only one accessible replica. Second, writing succeeds if one replica is accessible and enough spare nodes and disk space are available for regeneration. Third, restoring the full complement of replicas decreases resource vulnerability to multiple failures. In section 4.4, we will analyze and compare the availability offered by the multiple-copy update methods described in section 3.2.

Regeneration also adapts the replicated resources to system configuration changes. The above conceptual algorithm (figure 4.1) uses a 'lazy' regeneration strategy, in which a resource is restored to its full complement only when it is being updated. In systems where the configuration changes frequently, compared with the frequency of writes to all objects, some resources might dwindle and disappear before regeneration occurs. To avoid this, one could create a 'resource restoration' operation which would go through the directory, identify resources some of whose replicas have become inaccessible, and replace the lost replicas by new copies of the surviving replicas.

However, there is a price to the added availability provided by Regeneration. To restore inaccessible copies, Regeneration consumes spare disk space on accessible nodes with failure modes independent from the down ones. For example, if a maximum of two nodes and 100 megabytes of data are expected to be down at any given moment, we need two independent, accessible nodes with 100 megabytes of spare disk space to assure successful regeneration of all down data. In general, Regeneration works by moving inaccessible data out of down nodes, so the "extra" resource requirements are no more than the amount of expected unavailable hardware.

Another resource management problem is garbage collection. The out-of-date copies in the down nodes become garbage because they have been replaced. Methods to reclaim the disk space, such as garbage collection and queued delete messages, are summarized in section 4.2.3.

Table 4.1 compares Regeneration to Voting (section 3.2.2) and Available Copies (section 3.2.3). The first two rows show the number of copies accessed for read and write operations. The last row compares actions taken on the inaccessible copies to maintain resource consistency .

4.1.4 Algorithm Verification

Since the Regeneration algorithms are enclosed in transactions, the one-copy semantics of replicated resources are guaranteed. All read requests and write requests are serialized by the transaction mechanism, and write requests update all copies atomically, when they succeed. If they abort, all copies are left in their original state.

Read:

1. Find current replicas from the directory.
2. Send the *Read* request to successive replicas until one services the request.
3. **if** all replicas fail

   then return abort

   else return success

end if

Write:

1. Find current replicas from the directory.
2. Send the *Write* request to all replicas.
3. **if** the request failed on all replicas, **then return** abort

elseif the request succeeded on some but not all replicas,

   then 3.1. Find available nodes with failure modes independent from machines holding existing copies

   3.2. Make new replicas to reach the specified number

   3.3. **if** (3.1. or 3.2.) not successful **then return** abort

   else return the result from

   Write(new configuration) to parent directory

   end if

else { Request succeeded in all replicas. }

   return success

end if

Figure 4.1: The Conceptual Algorithm

update method ⇒ operation	*Voting*	*Available Copies*	*Regeneration*
Read	Read Quorum	1	1
Write	Write Quorum	all copies	all copies
inaccessible copies	ignore	exclude	replace

Table 4.1: Comparison of Read/Write Actions

Although transactions assure resource consistency, we are still interested in finding out under which conditions the Regeneration succeeds in reading or writing a resource. Theorem 1, below, shows that a resource replicated with Regeneration needs only $k+1$ copies to be k-crash-resistant for read access. Theorem 2 says that if enough spare nodes with disk space are available, then the same resource would be k-crash-resistant for write access. The number of spare nodes necessary is equal to the number of copies lost since last regeneration.

We define some terms before the theorems are stated.

Definitions: For a k-crash-resistant resource, we define:

$Copies \doteq$ The set of copies.
$ActiveCopies \doteq$ The subset of $Copies$ that is accessible.
$FailedCopies \doteq$ The subset of $Copies$ that is not accessible.
$ActiveCopiesNodes \doteq$ The set of nodes on which $ActiveCopies$ reside.
$TotalNodes \doteq$ The set of all independent nodes in the network.
$FailedNodes \doteq$ The set of nodes which are not accessible.
$SpareNodes \doteq$The set of accessible nodes without a copy of the resource, but with enough storage to accept a new copy.

Immediate relationships from Definitions:

$ActiveCopies \subseteq Copies$.
$FailedCopies \subseteq Copies$.
$ActiveCopies \cup FailedCopies = Copies$.
$ActiveCopies \cap FailedCopies = \emptyset$.
$SpareNodes = (TotalNodes - FailedNodes) - ActiveCopiesNodes$.

Theorem 1 *For a k-crash-resistant resource with $k+1$ copies, if $|FailedCopies| \leq k$, then a read request succeeds.*

Proof: First we calculate the number of accessible copies after k copies have been lost.

1. Find current replicas from the directory.

2. Send the *Write* request to every $c \in Copies$.

3. **if** the request failed on all replicas, **then** abort

   { Inv.1: $|ActiveCopies| \geq 1$.}

   elseif request succeeded in some but not all replicas

   -- case $|ActiveCopies| < k + 1$.

   then repeat

   { Inv.2: $|SpareNodes| + |ActiveCopies| \geq k + 1$.}

   Take $n' \in SpareNodes$,

   -- initially $k \leq |SpareNodes|$ so this succeeds up to k times.

   Make copy c' on n',

   such that $content(c') = content(c).\ c \in ActiveCopies$.

   $\{ActiveCopies \Longleftarrow ActiveCopies \cup \{c'\}\}$

   $\{SpareNodes \Longleftarrow SpareNodes - \{n'\}\}$

   until { Inv.3: $|ActiveCopies| = k + 1$.}

   Write(new configuration) to directory.

   endif

Figure 4.2: Write Algorithm with Invariants

$$
\begin{array}{lll}
|ActiveCopies| & = |Copies| - |FailedCopies| & \text{definitions} \\
& = (k+1) - |FailedCopies| & \text{hypothesis} \\
& \geq (k+1) - k & \text{hypothesis} \\
& \geq 1. &
\end{array}
$$

Therefore, at least one copy remains accessible after k copies have been lost. Since Regeneration tries all copies, the read request succeeds when the accessible copy is found.

Theorem 2 *For a k-crash-resistant resource with $k+1$ copies, if $|FailedCopies| \leq k \leq |SpareNodes|$ and enough disk space is available on $SpareNodes$, then a write request succeeds.*

Proof: First, we rewrite the algorithm, inserting some invariants (in curly brackets). Then we show that the invariants hold under the assumptions.

Figure 4.2 contains the write algorithm with the invariants in which we are interested. The first assertion near the beginning, { Inv.1: $|ActiveCopies| \geq 1$}, is true if the request did not fail on all replicas:

$$
\begin{array}{rll}
|Copies| > k \geq |FailedCopies| & & \text{hypothesis} \\
|ActiveCopies| & = |Copies| - |FailedCopies| & \text{definitions} \\
& \geq (k+1) - k & \text{combining} \\
& \geq 1. &
\end{array}
$$

In other words, since not all copies have failed, there is one that does succeed.

The second assertion, { Inv.2: $|SpareNodes| + |ActiveCopies| \geq k + 1$.}, is true at the beginning of the repeat loop:

$$
\begin{array}{rll}
|SpareNodes| & \geq k & \text{hypothesis} \\
|ActiveCopies| & \geq 1 & \text{the first assertion} \\
|SpareNodes| + |ActiveCopies| & \geq k + 1. & \text{adding two inequalities.}
\end{array}
$$

Through each iteration in the loop, $|SpareNodes|$ decreases by one, and $|ActiveCopies|$ increases by one. Hence, the sum remains constant and Inv.2 is maintained.

The third assertion at the end of repeat loop, { Inv.3: $|ActiveCopies| = k + 1$}, will become true at some time, because at each iteration we create a new copy, increasing the set *ActiveCopies*. Once the set *ActiveCopies* reaches $k + 1$ copies, it substitutes the old *Copies* set in the directory and the regeneration completes.

4.2 Making Regeneration Practical

For simplicity, we have excluded three non-trivial problems from the conceptual algorithm: network partitions, the reconfiguration of the root directory, and garbage collection. We shall consider each in turn.

4.2.1 Network Partitions

Network partitions happen when communication links fail, but the nodes continue to function. In each partition, the Regeneration algorithm, as stated, would regenerate independently, even when as few as one replica is accessible. This is the optimistic approach to network partitions, and we will have to rely on merging algorithms (some of which have been mentioned in section 3.3.2) to restore consistency after merging the partitions.

The alternative, pessimistic approach (section 3.3.1), would alter the Regeneration update slightly. Suppose we allow updates only in the majority partition, defined as the partition with the majority of nodes. Then, in the write algorithm described in figure 4.1, a conditional statement should be added before step 1: if the update is in the majority partition, proceed as usual, otherwise, abort. The same alteration works with any criterion defining the distinguished partition.

Finally, there are networks where partitions are unlikely, such as local area networks. In these environments Regeneration can be applied directly; Eden offers such an example.

4.2.2 Root Directory

Although the partition problem may be avoided in some situations, the root directory problem cannot. To use Regeneration in the implementation of crash-resistant resources, we need a repli-

cated root directory that can be updated. However, a replicated directory that can be updated is not necessarily reconfigurable (allowing changes to the number or location of copies). If we want to replicate the root itself using Regeneration, it has to be reconfigurable.

In section 3.4, we have discussed three solutions to the location of root directory in the implementation of crash-resistant resources –broadcast, full redundancy, and fixed configuration. Of these three, broadcast and full redundancy allow reconfiguration. Fixed-configuration, as its name indicates, fixes the configuration of the root. We shall consider each alternative in turn.

Broadcast allows reconfiguration of the root directory, so it may be implemented with Regeneration. If parts of the root directory become inaccessible and are replaced during a regeneration, broadcast will find the new copies of the root directory.

Full redundancy eliminates the root reconfiguration problem altogether. Updating the root directory writes on all accessible copies. The inaccessible ones are brought up-to-date when they join the network. So, a fully redundant root directory requires a recovery mechanism independent of the multiple-copy update method used in the replication of other resources.

Fixed configuration restricts the root reconfiguration. So, although crash-resistant resources may be implemented with Regeneration in a system with a fixed set of copies for the root, the root should be replicated using some other technique, for example, Available Copies. Otherwise the update of the root would fail if one (or more) of its copies is inaccessible.

In our experimental implementation, ultimately we rely on Eden kernel's broadcast to find our root directory replicated with a different technique, to be described in section 5.2.1.

4.2.3 Garbage Collection

Regeneration replaces inaccessible copies with new ones on accessible nodes. Therefore, when the down nodes recover, we need to reclaim the disk space occupied by the old copies. This problem is relevant to resource management, and does not cause inconsistency in resources replicated with Regeneration.

There are two ways to recover the disk space. The first is a system garbage collector, which eliminates the problem. For example, our implementation in Eden, described in chapter 5, relies on the Eden garbage collector. However, garbage collection in distributed systems remains a research problem [47], so this option may be unavailable in some systems.

The second alternative is to queue delete messages in some other nodes, to be delivered when the down nodes recover. The copies are then deleted and the disk space released. As long as the replicas have their own unique identifiers, each generation is distinct from the predecessors and the messages will delete only the intended copies. There may be some delay in the delivery of queued messages because the nodes holding the messages may be down or busy. These situations may slow down disk space reclamation. If the problem becomes severe, messages may be replicated to decrease the probability of delay to any desired level.

4.3 Comparison with Previous Work

The most important idea in Regeneration is the separation of data restoration from hardware repair. Other replication methods, in particular those described in section 3.2, can adopt the idea,

but to the the best of our knowledge, this separation has not been reported before. In section 4.4, we will show, using a probabilistic analysis, that this separation increases resource availability. In this section, we compare the algorithmic aspects of Regeneration with other multiple-copy update techniques.

4.3.1 Primary Copy

With a fixed primary site, a resource replicated with the Primary Copy method increases read availability only, through secondary copies; its write availability is the same as the availability of the primary site. In contrast, Regeneration and the other methods in this section all provide some way of increasing write availability. Also, unlike Primary Copy, which is an asymmetric replication method because of the distinguished primary copy, Regeneration and the other methods are symmetric, i.e., all copies are equal.

For network partitions, Primary Copy solves the problem by imposing the pessimistic approach, restricting the updates to the partition containing the primary copy. Regeneration, as stated in section 4.1, requires some other technique to handle partitions, which may be either optimistic or pessimistic.

Primary copy may use the idea of regeneration with respect to secondary copies. During an update, if some secondary copies are inaccessible, other secondary copies can be created. However, regeneration of the primary copy is not easy. For example, Minoura's True-Copy Token scheme [59] requires that we certify all true copy tokens as lost before a regeneration can take place.

4.3.2 Majority Voting

Regeneration, like Available Copies, reads one and writes all copies, while Majority Voting reads and writes subsets of copies. Because of the read/write trade-off in Voting, to preserve fault-tolerance in write, the read quorum must be at least two. Consequently, if remote accesses are more expensive than local access, Voting would impose higher access costs. In compensation, Regeneration requires spare nodes and storage to regenerate new copies, and space occupied by replaced copies needs to be reclaimed.

Voting and Regeneration also differ in their handling of network partitions. Regeneration needs some additional method to guarantee resource consistency, while Voting imposes the pessimistic approach by the selection of a majority quorum for updates.

Voting can use the idea of regeneration to increase resource availability. For example, in gathering a write quorum, the quorum collector can copy the current version onto obsolete representatives [36]. What we suggest is that in the absence of partitions, and given the flexibility to update configuration data, new replicas can be created on spare nodes to obtain either a read or a write quorum.

4.3.3 Available Copies

Regeneration and Available Copies share three characteristics. First, they are both multiple-copy udpate algorithms based on reconfiguration. A directory is used to indicate which copies are

up-to-date. Second, both algorithms read one and write all copies. Finally, they are both "pure" multiple-copy update algorithms, which allow the adoption of optimistic approaches to handle network partitions. If used in a network in which partitions are likely, they must be combined with some network partition handling technique.

As with other multiple-copy update methods, Regeneration differs from Available Copies because of the replacement of inaccessible copies. In addition, there are two important differences between them. The first is the amount of work done at crash detection and recovery. The Available Copies algorithm is pessimistic, excluding all potentially out-of-date copies, while Regeneration is optimistic, replacing only actually inconsistent copies.

The second difference lies in the scope of adaptation: Available Copies may exclude out-of-date copies, which remain inaccessible until their nodes recover. In contrast, Regeneration can migrate copies out of retired nodes into new hardware. However, Regeneration requires more sophisticated resource management: spare nodes and storage are required for regeneration, and space occupied by replaced copies needs to be reclaimed.

Since the Available Copies algorithm requires the capability for reconfiguration, it is easy to add regeneration to resources replicated with Available Copies. DDM [23] uses Available Copies, and it includes the concept of offline copies that can be switched online. Although the offline copies are more like hardware cold stand-bys than regeneration, the final result is equivalent to some degree. The offline copies decrease resource vulnerability to multiple failures, up to the number of offline copies. Regeneration is free from the limitation of the number or location of offline copies.

4.4 Availability Analysis

In section 4.3, we have compared the algorithmic differences of several multiple-copy update methods. Now we proceed to compare the availability they provide, using a probabilistic analysis. Four replication techniques, Primary Copy (PC), Voting (VT), Available Copies (AC), and Regeneration (RG) are analyzed and compared in this section.

Performance characteristics like throughput or response time are beyond the scope of this analysis, concerned with availability. The cost of replication, in terms of disk storage space, is fixed in the analysis, since we use the same number of copies to allow a fair comparison. With a fixed number of copies, the disk space required would be the same for all methods. Execution overhead depends on other factors such as the read/write mix, and impacts the overall performance. Integrated performance and availability analysis remains a subject of future research.

4.4.1 The k-out-of-N Model

The availability of replicated resources can be analyzed using results in reliability theory on k-out-of-N systems [4]. The assumptions of the k-out-of-N model are:

1. The resource and its copies are 2-state: either accessible or inaccessible.
2. State changes of the copies are statistically independent.
3. The resource is accessible if and only if at least k of its N copies are accessible.

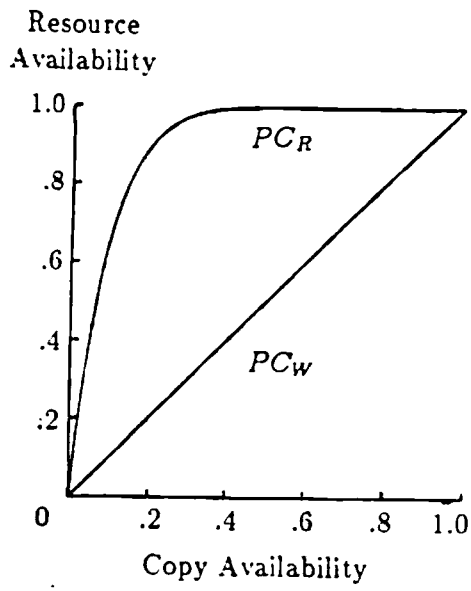

Figure 4.3: Primary Copy

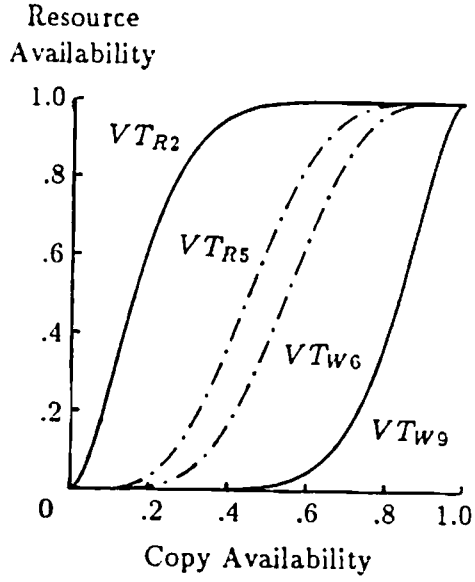

Figure 4.4: Voting

The main result of k-out-of-N model we will use is that, given the availability of each copy, we can combine all possible system configurations to calculate the availability of the resource. Our analysis will be illustrated by a series of diagrams showing the resource availability as a function of copy availability. In these diagrams, all copies are assumed to have the same availability, which appears on the horizontal axis; the curves show how the resource availability improves when the copy availability covers the probability range from 0 to 1.

For concreteness, we study the availability of a resource with 10 copies. The analysis applies to systems in which:

- Nodes are either up or down (no Byzantine failures; e.g. fail-stop processors [76]).
- Copies reside on independent nodes.
- Nodes have same probability of survival (e.g. in a homogeneous network).

4.4.2 Primary Copy

A replicated resource with 10 copies using the Primary Copy method allows writes on the primary copy and reads on any of 10 copies. Consequently, its write availability is the same as the primary copy, shown in figure 4.3 by the straight line (PC_W). Its read availability is much higher, since any one out of *10* copies would do; this is depicted by the left curve (PC_R) in the same figure.

4.4.3 Majority Voting

For 10 copies, simple Majority Voting allows 8 choices of read and write quorums, from (read 2, write 9) to (read 9, write 2). Of the 8 choices, the 4 in table 4.2 are useful when the transaction reads the resource before the update. We will consider two representative examples:

Number of Copies		curves in
Read Quorum	Write Quorum	figure 4.4
2	9	VT_{R2}, VT_{W9}
3	8	omitted
4	7	omitted
5	6	VT_{R5}, VT_{W6}

Table 4.2: Majority Voting, Quorum Choices for 10 Copies

1. The solid lines in figure 4.4 show the *2*-out-of-*10* read availability (VT_{R2}) and *9*-out-of-*10* write availability (VT_{W9}) of a read quorum of 2 and write quorum of 9. In this case, the resource read availability is high, but write availability is correspondingly low, demonstrating the read/write availability trade-off in Voting.

2. A read quorum of 5 and write quorum of 6, is shown in figure 4.4 by broken lines (*5*-out-of-*10* and *6*-out-of-*10*, marked VT_{R5} and VT_{W6}, respectively). In this case, both read and write availability are higher than VT_{W9}, but lower than VT_{R2}.

4.4.4 Available Copies and Regeneration

Given 10 copies, Available Copies offers the (*1*-out-of-*10*) read and write availability, shown in figure 4.5. Assuming there are enough resources to allow successful regeneration, the Regeneration method provides the same (*1*-out-of-*10*) read and write availability.

The difference between Available Copies and Regeneration arises after a successful regeneration. Let us consider the situation where a number of copies have been lost, for example, five. Out of the remaining five, Available Copies provides *1*-out-of-*5* availability, the right curve (RG_B/AC) in figure 4.6. With five remaining copies, Regeneration also provides *1*-out-of-*5* availability. However, after successful regeneration, the resource availability is restored to *1*-out-of-*10* (RG_A). Figure 4.6 shows the availability increased by regeneration, which the Available Copies algorithm does not provide. In the figure, a 10-copy resource's read and write availability before the regeneration (RG_B/AC with 5 accessible copies) is compared to its availability after regeneration (RG_A with 10 accessible copies).

4.4.5 Comparison of The Four Methods

We can compare the four methods, Primary Copy, Voting, Available Copies, and Regeneration, by superimposing the figures 4.3, 4.4, 4.5 to obtain figure 4.7. The solid curve to the left, representing read and write availability provided by Available Copies and Regeneration (and Primary Copy

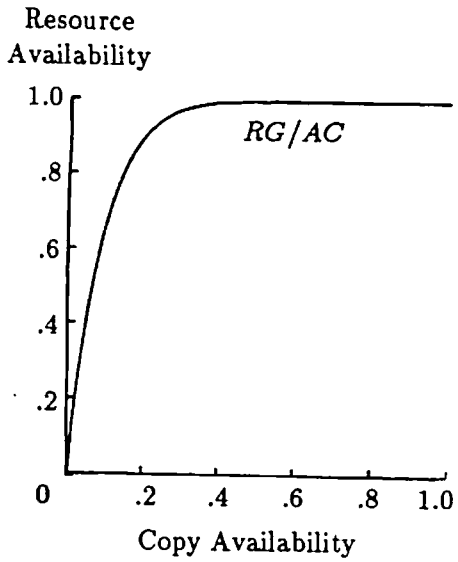

Figure 4.5: Available Copies and Regeneration

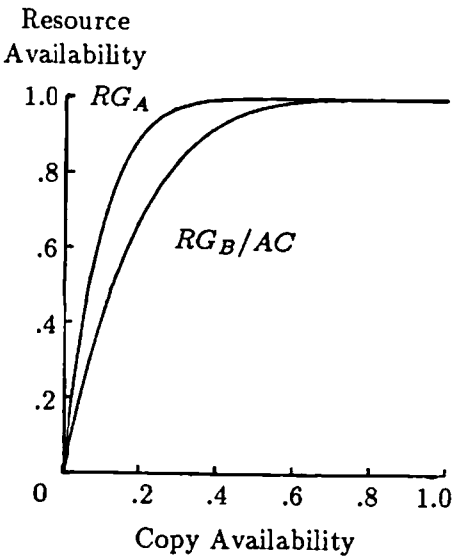

Figure 4.6: Regeneration, before and after

read-only), is higher than the whole range of availability covered by the Voting quorum choices, exemplified by four dot-dash lines. Also note that although VT lines show high resource availability at high copy availability, they all cross the straight line, representing non-replicated resource (and Primary Copy write) availability. This means that for sufficiently low copy availability levels, Voting actually offers less availability than a non-replicated resource.

In order to clarify this point and examine different replication techniques performing "under stress," let us consider a 10-copy resource of which 5 copies are not accessible. Of the figures considered so far, 4.3, 4.4, 4.5, and 4.7 have been calculated based on the initial resource configuration, where 10 copies have been created and are accessible. Figure 4.8 shows the resource availability provided by the four methods, when only 5 copies remain accessible.

First, we observe that Voting write availability for either a write quorum of 6 or 9 is zero, since such quorums cannot be gathered with only 5 copies. This offers an intuitive explanation for the low Voting availability at low levels of copy availability: the copies are more likely to be down, making the vote gathering less likely to succeed.

Second, the dotted line (PC_W) represents the primary copy availability *if* it is among the 5 accessible copies. Otherwise the primary copy is simply inaccessible. The reason PC_W remains constant is that the availability of a non-replicated resource is always the same as the availability of the node on which it resides.

Third, the solid curves (from right to left) show the *5*-out-of-*5* read availability (VT_{R5}) from a read quorum of 5, the *2*-out-of-*5* read availability (VT_{R2}) from a read quorum of 2, and *1*-out-of-*5* read and write availability ($AC/RG_B/PC_R$) of Available Copies and Regeneration (Primary Copy read-only). It is relatively easy to see that the *5*-out-of-*5* curve drops quickly as the copy availability drops a little. The *2*-out-of-*5* curve is intermediate, while *1*-out-of-*5* is the highest of the three solid lines.

Finally, the dashed curve represents the *1*-out-of-*10* read and write availability (RG_A) offered

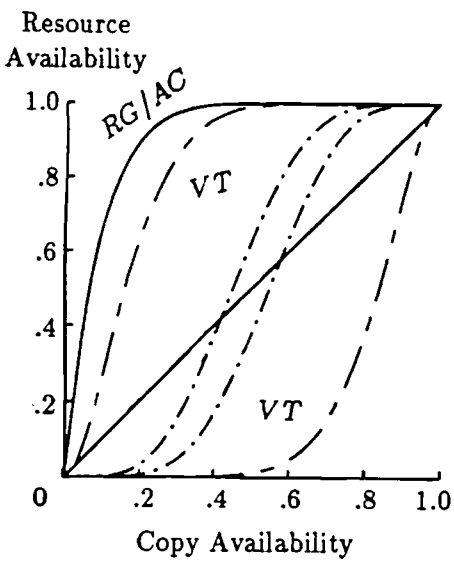

Figure 4.7: Comparison, 10 copies

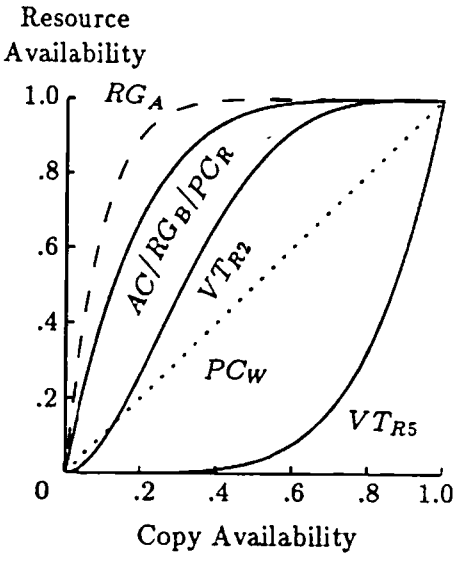

Figure 4.8: Comparison, 5 remaining copies

by Regeneration, assuming enough nodes allowed successful regeneration. The *1*-out-of-*10* curve is clearly higher than all other curves, showing the availability increase due to regeneration.

So, system degradation decreases resource availability for all four methods, but Regeneration is the only one to repair the resources even before the nodes recover. In order to take advantage of this feature, if Regeneration is used in an application where the ratio of writes to reads is very low, one should force regeneration often enough to prevent loss of the last copy. There are two ways this can be implemented. First, the read algorithm in figure 4.1 may be modified to perform a regeneration whenever a copy (or a certain number of crashes) is found inaccessible. Second, a daemon can run in the background checking for inaccessible copies. The daemon could run from time to time, or after the detection of a node crash (or a certain number of them) in the network.

Regeneration has two strong points:

1. Increased availability allows fewer initial copies, reducing disk space requirements.
2. The ability to adapt resources in a changing environment permits more flexibility in system configuration.

However, regeneration may fail for insufficient disk space on spare nodes. In this case, we could regenerate up to the number of operable nodes, instead of aborting the update. This can be viewed either as the Available Copies method with regeneration added, or as the Regeneration method with a variable number of required copies. The optimal number of copies is a function of the storage cost, the execution overhead, the probability of crashes, and the frequency of regeneration. One should regenerate up to a replication level that provides an acceptable availability of the resource, and not beyond.

Chapter 5

Replicated Resource Distributed Database

5.1 Design

We have implemented the Replicated Resource Distributed Database (R2D2) to supply a replicated directory to the nested transactions mechanism, described in chapter 6, which supports crash-resistant resources. R2D2 is a crash-resistant mapping of string names into sets of capabilities. Using the Regeneration method for replication, R2D2 also demonstrates the practicality of the Regeneration method.

In section 5.1.1, we describe the client interface to R2D2. Since R2D2 is built with Eden objects, we introduce some internal structures of Eden objects in section 5.1.2. In section 5.1.3, we summarize the Core Structure in which the directory mapping of R2D2 is replicated. The one-copy syntax and semantics are assured by the Access Structure, summarized in section 5.1.4.

5.1.1 R2D2 Client Interface

R2D2 is an atomic data type [56,83], in the sense that each invocation to R2D2 is atomic despite concurrent access and system crashes. The main R2D2 operations (invocations) access the mapping of resource names into sets of capabilities:

- LookupSet(in: StringName; out: CapaSet, Status)
  - returns the set of capabilities named by string name.
- AddSet(in: StringName, CapaSet; out: Status)
  - add the pair (resource name, capability set) into the mapping.
- DeleteSet(in: StringName; out: Status)
  - delete the entry named from the mapping.
- ReplaceSet(in: StringName, CapaSet; out: Status)
  - replace the named mapping entry with a new capability set.

These operations are part of the abstract type RepDirectory, defined in appendix table A.1.

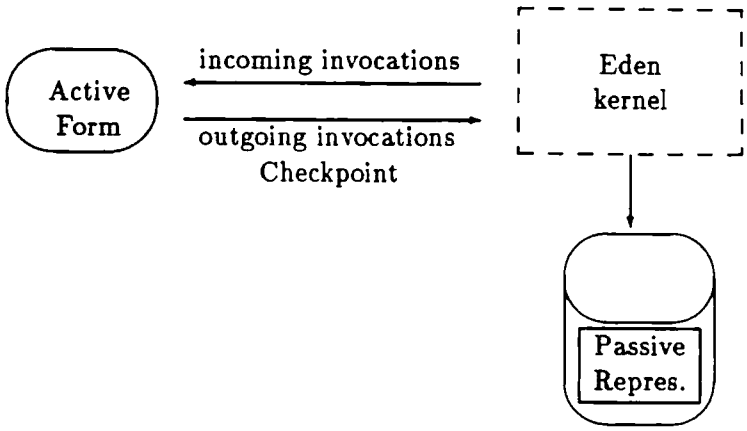

Figure 5.1: Eden Object's Two Forms

R2D2 is also a crash-resistant resource. Components of R2D2 are replicated using Regeneration. The degree of replication (the number of copies) varies from component to component, and can be changed any time. The replication in R2D2 is completely transparent to its clients. All requests are sent to R2D2Root, and the internal structures described in this chapter are hidden.

5.1.2 Eden Objects

In section 1.3 we have outlined the main concepts of the Eden system. Now we describe the current implementation of Eden, making these concepts more concrete.

The most important characteristic of Eden objects is the distinction between an object's active form and passive representation (see figure 5.1). An object's active form is a Unix process running the object's Edentype code, on a node with an Eden host. Invocations delivered by the host to that object are serviced by its active form. The active form can decide to checkpoint, causing the object's entire state to be written into the passive representation, on stable storage. The checkpoint replaces an earlier version of the long-term state atomically; either the old version or the new version is accessible to the object at all times.

If the active form crashes, the Unix process goes away, and only the passive representation remains. Eden kernel is capable of reactivating the object, given the passive representation. For example, invoking an inactive object causes it to be reactivated by the kernel. Since the checkpoint is an atomic operation, there is always a consistent version of the passive representation.

We have mentioned that each object is an instance of an Edentype, a program written in Eden Programming Language (EPL) [16]. EPL is an extension of Concurrent Euclid, which is a Pascal derivative. The fundamental unit of program in Concurrent Euclid is a *module*. Modules may be put together to form larger programs, and there are well-defined rules for information sharing and hiding between modules. Basically, data structures are internal to a module and cannot be shared across modules, but operations can be invoked from another module. In this aspect, modules are similar to instances of abstract data types.

Each module may have several *processes* to express multiple threads of control. A process starts at the beginning of program execution, and ends when executable commands run out. The program exits when all of its processes end. A process may stop in the middle of its execution, however. Processes may wait on condition variables for certain events to occur. These condition variables, protected by monitors, constitute the basis of synchronization between processes.

The original Concurrent Euclid, as defined by Holt [43], provides only static declaration of processes. The Concurrent Euclid compiler used in the Eden project, enhanced by Norm Hutchinson, supports dynamic creation of processes. At run time, a declared process may fork as many processes as necessary. All processes are equal, since there is no child/parent relationships between forked and declared processes.

Processes are used in two ways: First, active processes do real work. For example, Eden objects that interact with human users have a process that collects keyboard input. Second, passive processes service invocation procedures. They wait for invocation messages to arrive, and then execute the especially designated "invocation procedure" for that invocation.

To illustrate the structure of an Eden object, a simple Edentype, the *EdenInteger* has been written and its EPL program included in appendix section A.2.5. Although EdenInteger has only passive processes, the active ones are syntactically and semantically the same as the passive processes declared to service invocations.

R2D2 is implemented with Eden objects more complex than EdenInteger, but the basic structure and language primitives are the same. In the rest of this chapter, we describe the R2D2 components in terms of their function. More implementation details of the Edentypes involved can be found in appendix section A.2.

5.1.3 Core Structure

R2D2's core structure is a replicated tree (figure 5.2 shows the tree's top portion). Each box in the figure represents an object of Edentype RepDir, a mapping of strings into sets of capabilities, which point to replicas at the next level in the tree. The double arrows denote the sets of capabilities for the replicated resources in the mapping. For example, the root of the tree in figure 5.2 is called '/'. The mapping in the root has two entries, one with name 'users', and the other 'system'. Both 'users' and 'system' have two copies, so each name maps into two capabilities.

An instance of RepDir is unaware of its replicas (shadow boxes in the figure). For example, it does not contain any concurrency control. Therefore, RepDir is not an atomic data type. The atomicity of R2D2 operations are assured by the transaction managers in the Access Structure (section 5.1.4). RepDir's only salient feature is the idempotency of all write operations (Add, Delete, and Replace). Section 5.2.3 describes the implementation of RepDir, and more details are in appendix section A.2.4.

5.1.4 Access Structure

In order to keep the replicas in the core structure consistent, we have an access structure on top of the core (figure 5.3), composed of an R2D2 Root, and several identical instances of the R2D2 Transaction Manager (R2D2TM). Each R2D2 request is received by the Root, and forwarded to

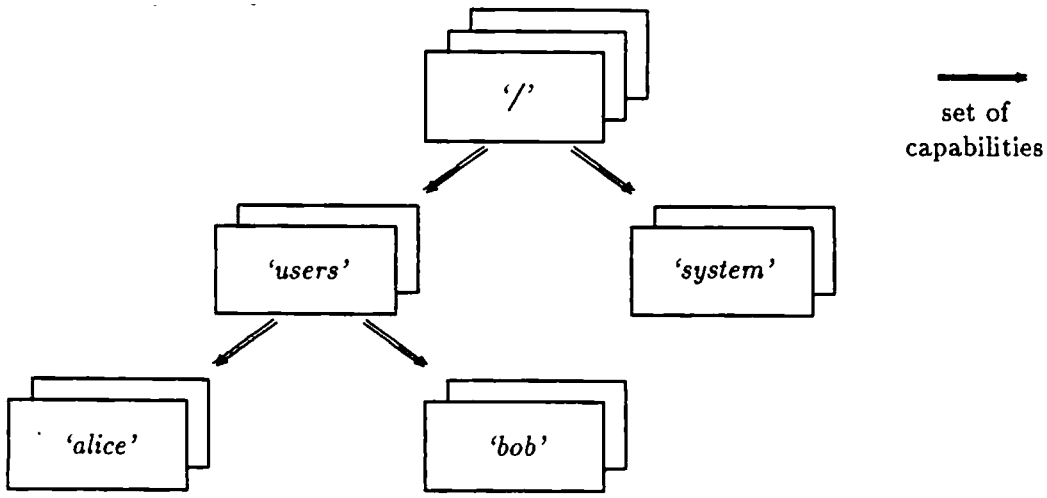

Figure 5.2: Core Structure - A Tree Structured Mapping

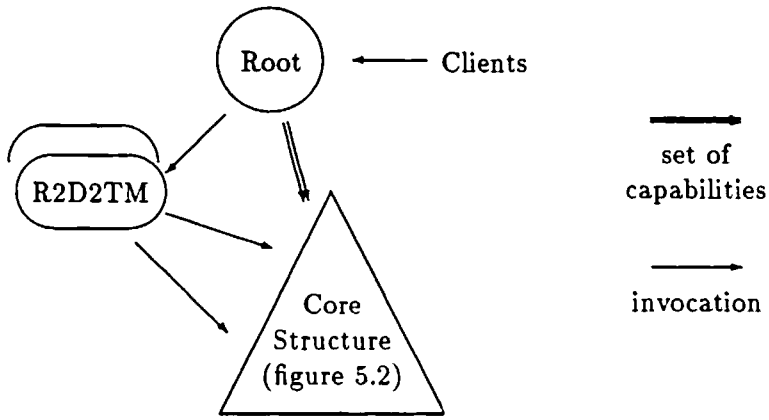

Figure 5.3: Access Structure - On Top of Core

an idle R2D2TM. The main function of an R2D2TM is to keep replicas in the core structure consistent during an update, using the Regeneration method.

The Root forwards the invocations to R2D2TM for two reasons: separation of functions, and load distribution. R2D2TMs can be seen as local or remote processes "forked" by the Root to take care of individual requests. At the time the decision was made, we felt that Eden should be used as a tool for distribution. From this point of view, the R2D2TMs take advantage of transparent distribution of Eden objects to offload work from the Root. However, the attempts to measure the load balancing effects of distributed R2D2TMs met practical difficulties. Since the objective of our study is replication for availability, not efficiency, those measurements have not been completed.

Separating the Root and R2D2TMs into individual objects seemed natural at the time the decision was made, because of potential load balancing benefits. However, the immediate consequence has been that each R2D2 request is handled by an intermediary object (R2D2TM), whose

only job is to maintain concurrency and reliability atomicity of R2D2 as a crash-resistant atomic data type. The introduction of an intermediary adds two Eden invocations in the processing of each R2D2 request. Since Eden invocations are expensive, the resulting impact on R2D2 performance, as compared to the non-replicated case, became noticeable. In section 5.3.4, we will discuss this further.

5.2 Implemenentation

The implementation of R2D2 reflects our analysis of crash-resistant resources. The Root, described in section 5.2.1, provides the one-copy resource access syntax. The R2D2TM, described in section 5.2.2, assures the integrity of one-copy resource access semantics. Finally, the RepDir, described in section 5.2.3, replicates the data.

The design of R2D2 proceeded concurrently with the design of ERMS (chapter 6) and the development of the Regeneration method, taking about one man-year. Coding and debugging of R2D2 took slightly over three man-months.

5.2.1 R2D2 Root

The Root serves two functions: transparent access to the replicated core structure and lock management. Since the Root has a fixed capability, clients use one single capability to address R2D2. However, the fixed capability prevents the use of the Regeneration method to replicate Root data at the object level. Nevertheless, the Root's data must be replicated somehow, since it participates in every access to the replicated database. We solve this problem by making the Root a *replect* [65,72], a special kind of *repl*icated ob*ject* that we now describe briefly.

A replect's passive representation is replicated on several disks, using a variant of Gifford's Voting scheme, and is implemented within the kernel, rather than at the object level. There is only one active form for each replect at any time. If the active form crashes, and should enough replicas of passive representation be accessible, the next invocation automatically reactivates the replect on another node.

The main advantage of a replect is transparent access, since it is invoked with a single capability, in exactly the same way as non-replicated objects. However, to achieve a given level of availability, a replect needs more copies of its data than an object replicated by the Regeneration method (see analysis in section 4.4). The transparent access was instrumental in the decision to adopt the replect for R2D2Root.

To implement the Root directory, there are inherent difficulties with the other alternatives discussed in section 3.4. Broadcast was used at the Eden kernel level, but it was not available to Eden object programmers as a primitive. In other words, Eden objects cannot send a message to a group, or all of the objects in the system. Full redundancy was also unavailable to object programmers, since each capability is mapped by the kernel into only one object. So there is no manner in which a specific capability can address several objects residing on different nodes. Finally, since we need only an updatable root, a fixed configuration could have worked, and would have been chosen by default. The implementation of replects by A. Proudfoot [72] provided a superior solution than fixed configuration for two reasons. First, only one capability is necessary

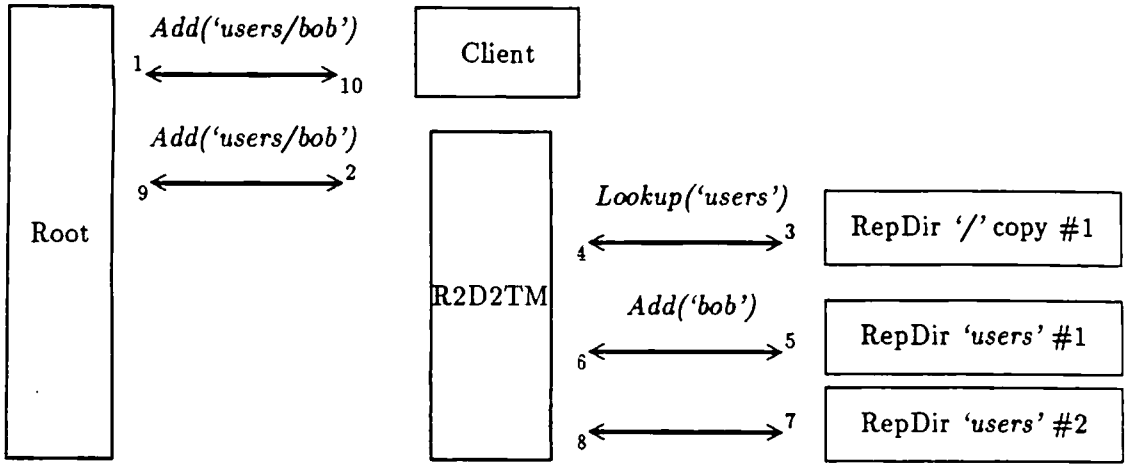

Figure 5.4: R2D2TM Actions in *Add('users/bob')*

to invoke the replect Root. Second, the replect provides some fault-tolerance for updating the Root.

The Root's lock table is a mapping from string names to the unique identifiers of the transactions holding the locks on the names. (More details of LockTable are in appendix section A.3.5). Currently, exclusive locks are used to serialize updates. Other concurrency control methods could have been used, but this simple approach was taken because the main focus of the work was replication rather than concurrency control. RepDirs have internal synchronization to ensure consistent read access, so no external read locks are necessary. In order to update an entry, the name corresponding to the parent mapping is locked. For example, *Delete('users/bob')* requires a lock on *'users'*.

5.2.2 R2D2 Transaction Manager

The R2D2TM receives a request from the Root, and keeps the core structure consistent. The way an R2D2TM works is best illustrated by the following example. Figure 5.4 shows the work done in R2D2 to service the invocation *Root.Add('users/bob')*; the actions are numbered in the order in which they happen and their numbers either appear above an arrow (an invocation) or below (a reply to an invocation).

Upon receipt of the client request (1), the Root finds an idle R2D2TM, and forwards the invocation (2). Since this is an update, the R2D2TM receives an implicit lock and proceeds to find the RepDirs containing the map entry (3). This search goes down the tree as many levels as necessary, and returns the capability set corresponding to the RepDir which is to be updated, in this case *'users'* (4). The real update invocations are now sent to all the replicas of *'users'* (5 and 7). If these are successful (6 and 8), then the R2D2TM returns the success status to Root (9), unlocking implicitly. Finally, Root forwards the result to the client (10). This no-crash scenario is the normal case. Cases involving crashes or inaccessible copies are analyzed in section 5.2.4.

5.2.3 Replicated Directory

The core structure objects (RepDirs) are essentially composed of two mappings. The first, DirMap (appendix section A.3.3) maps strings into sets of capabilities. The second, TIDMap (appendix section A.3.9), maps unique transaction identifiers (TIDs) associated with updates into their return status codes. Updates are made idempotent by searching the TIDMap first. If the update's TID is found, the corresponding status code is returned and no action taken. Otherwise the update is performed and the result status associated with the new TID.

The RepDir objects themselves are unaware of the replication of their data. Rather, the replication information is stored in their parents in the directory hierarchy, in the mapping of each string into a set of copies. Consequently, RepDirs can be seen as simple abstract machines that implement a mapping.

5.2.4 Regeneration in R2D2

There are several places where R2D2 may have to replace an inaccessible object by a working alternative. In the above example, figure 5.4, the first time this can happen is between actions 2 and 9. If the Root detects a crash of the R2D2TM servicing the request, the Root simply allocates another R2D2TM, sending it the same request with exactly the same TID. Since RepDir invocations are idempotent, the re-execution does no harm. This is a case where programs are replicated to increase fault-tolerance.

The second time an inaccessible object may be encountered is between actions 3 and 4. If one replica of the RepDir named '/' is not accessible, the R2D2TM will try the next replica. Since we are only reading, our "lazy" regeneration strategy bypasses regeneration. As long as one replica remains accessible in the chain, we continue going down the tree structure.

The third time an object may be replaced is between actions 5 and 8. For example, suppose that action 8 resulted in failure instead of success. Figure 5.5 describes the additional R2D2TM actions required by the Regeneration method to recover and proceed. The R2D2TM asks an up-to-date replica to make a copy of itself (8.1), and the capability of a new replica is returned (8.2). The R2D2TM finds a suitable new node (which does not currently hold a copy, and contains enough spare disk space) for the new copy, and makes sure the new replica stores its state on disk in the new node (8.3). When the new replica is securely established (8.4), the R2D2TM sends a request to Root (8.5), changing the mapping to reflect the new configuration. Note that now the R2D2TM assumes the role of client in figure 5.4, and another R2D2TM will be allocated to service the *Replace* request. When the configuration change is completed (8.6), the R2D2TM proceeds in its original course (figure 5.4, action 9).

The Root maps the distinguished resource name '/' into the top layer of the core structure. In the above example, the R2D2TM servicing *Replace('users')* may find a replica of '/' inaccessible. In this case, the root intercepts the resulting regeneration request *Replace('/')*, changes its mapping, and checkpoints the new mapping. This ends the recursion. Thus, every recoverable crash within R2D2 is invisible to clients, and although a crash of the Root cannot be hidden, its recovery consists of simply re-invoking the Root with exactly the same parameters. If the Root reactivation succeeds, R2D2 will start up again.

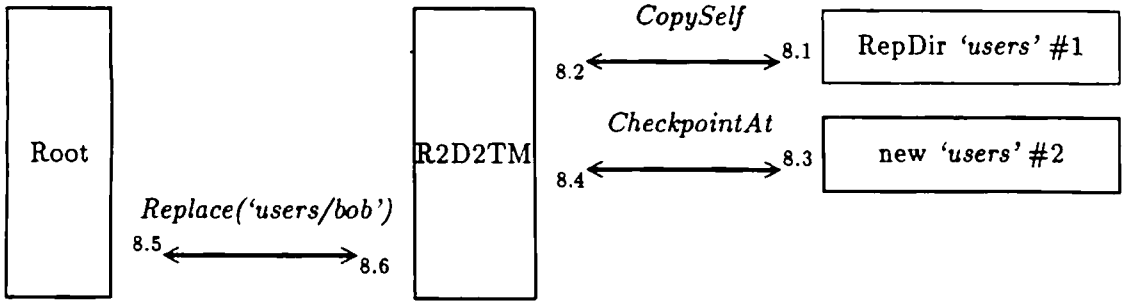

Figure 5.5: R2D2TM Replaces an Inaccessible RepDir

The client re-invocation is important, because a catastrophe might have crashed the Root and all R2D2TMs in service, leaving some RepDirs temporarily inconsistent. We rely on the clients to recover the consistency in R2D2 for three reasons:

1. In Eden, since a costly checkpoint is the only primitive to access stable storage (see figure 5.2 for some numbers), maintaining consistent data on stable storage is very expensive.
2. Client retry is easy (a simple loop) and in the normal case adds no extra cost.
3. The multi-workstation catastrophic crash is rare; since the Root and R2D2TMs are distributed, single-node crashes mean the survival of either the Root or the R2D2TM, and R2D2 consistency is assured.

Although R2D2 is replicated for availability, there are three cases in which R2D2 may be unable to service a request:

1. Every object in a capability set may be inaccessible (insufficient replication).
2. The Root activation may fail because of insufficient replect replication.
3. Regeneration may fail because of insufficient spare nodes or disk space (insufficient resources).

In the first case the resource will remain inaccessible until a node containing a replica recovers; in the second, the R2D2 will remain inaccessible until the Root replect recovers. In the third case, with insufficient resources, R2D2 may be unable to restore the full complement of copies, so regeneration cannot complete successfully. A modification of R2D2 to use variable number of copies may overcome this difficulty (section 4.4.5), although it remains to be implemented. Another approach is to use Regeneration in a system with enough resources so that this problem will not arise.

5.3 Measurements and Evaluation

5.3.1 Eden System

Eden has been running on an Ethernet of SUN workstations since Spring of 1984. The current hardware configuration includes 12 diskless workstations and 4 disk servers (figure 5.6). There

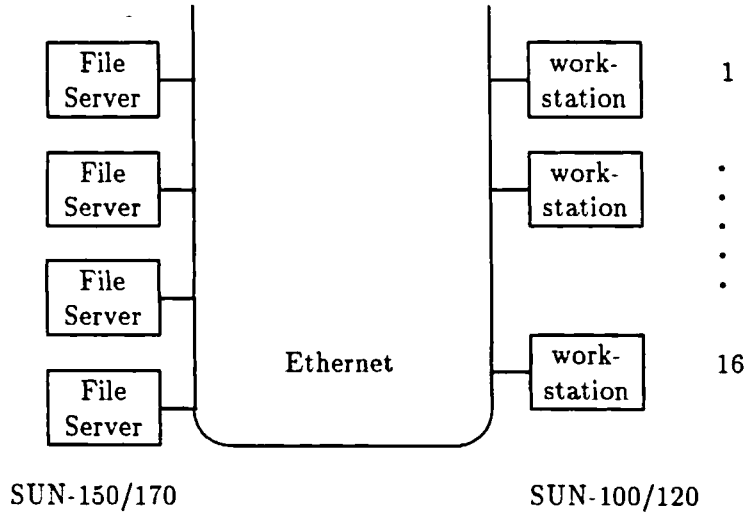

Figure 5.6: Eden Hardware (circa Spring 1985)

are two kinds of Eden kernels running as Unix processes on the SUNs, the host and the POD (Permanent Object Database). The host takes care of inter-object and inter-node communications, and the POD manages stable storage. An Eden object's active form runs also as a Unix process, and communicates with its Eden host through IPC calls. An object is always activated by "attaching" it to a host. All communications between the object and outside world (other objects and Eden kernel), is handled by the object's host.

PODs function as virtual back-end processors and they only talk to other PODs and hosts. There is always a "responsible POD" for each object's passive representation. When an object asks the Eden kernel to checkpoint its passive representation, the host receives the message and forwards the data to the responsible POD for processing. An object's passive representation data are stored in a Unix file. If the responsible POD runs on the same machine as the object's host, the POD is given the file. Otherwise the Eden Message Module transfers the file through Ethernet to the POD (see figure 5.7). Once in possession of the new passive representation file, the POD uses the atomic Berkeley Unix 4.2 command "rename" to switch from the old passive representation to the new one.

To an object programmer, the two most important Eden primitives are invocations and checkpoints. Invocations are the only way to communicate with other objects, and checkpoints are the only way to write to stable storage. Since R2D2 contains many objects, we use invocations. Of these objects, RepDirs need to store their state on stable storage, so they use checkpoints.

In the current prototype implementation of Eden, the costs of invocations and checkpoints are considerable. Eden timings indicate a *local* invocation between Eden objects on the same machine takes about 68 ms. Invoking an object on another machine (a *remote* invocation) takes about 103 ms. These numbers refer to an "empty" invocation, where there is no user processing, only the packing, unpacking, copying, sending and receiving of the messages. A checkpoint takes more than a second for a small amount of data (a few hundred bytes). Table 5.1 contains a compilation of available performance data on Eden primitives. We should note that the faster

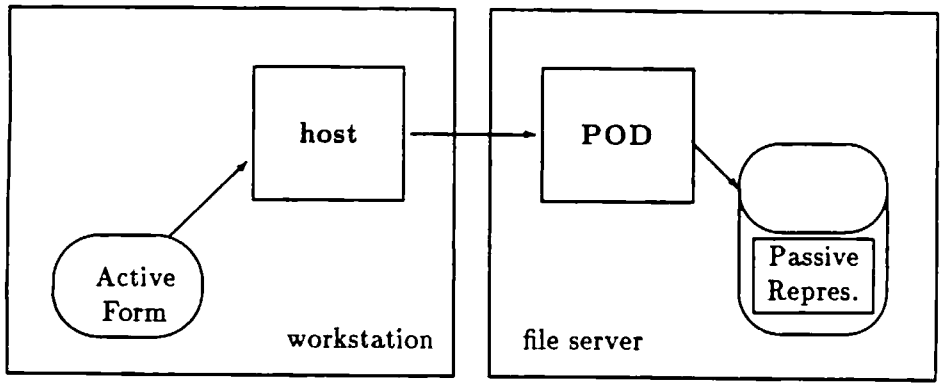

Figure 5.7: Host and POD

Operation	Local	Remote
Invocation *ReceiveAny*	0.056	0.084
Invocation *Normal Case*	0.068	0.103
Checkpoint *File Server*	0.9	—
Checkpoint *Workstation*	1.1	1.3

Table 5.1: Informal Eden Timings (in seconds, SUN 4.2)

timings, Invocation (ReceiveAny) and Checkpoint (file server), are restricted to specific cases. In R2D2, we needed and used Invocation (Normal Case) and Checkpoint (workstation). Since we only use these primitives, no further analysis of their cost will be given in this dissertation. Interested readers are referred to a Master's Thesis on invocation costs [44] and an internal study of checkpoints [57].

5.3.2 Experimental Set-Up

For our measurements, we have put the hosts on diskless workstations and PODs on disk servers. In other words, each object is activated on a diskless workstation, but its passive representation resides on a disk server. The above configuration is a natural one. It does not use disk access from diskless workstations, and uses disk servers only for stable storage. Other configurations are possible, using Eden hosts and PODs as logical nodes running on top of Unix. However, our goal

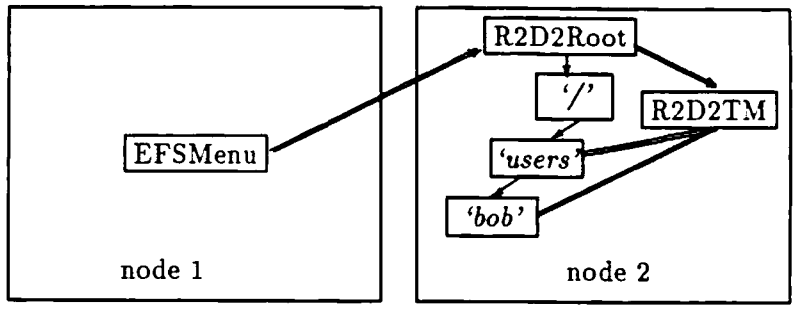

Figure 5.8: Reading R2D2

is to observe the performance of R2D2 as a crash-resistant resource, so we have used the same configuration in all measurements. Additional experiments with different configurations may shed light on Eden kernel implementation, but not on replication techniques.

In our measurements, R2D2Root is an ordinary Eden object, instead of a replect (described in section 5.2.1). There are several reasons for this simplification. First, the R2D2Root does not checkpoint during our measurements. Since the active form of a replect is indistinguishable from the active form of an object, the results must be the same. Second, the loss in availability does not affect these experiments, which measure only performance. Third, the implementation of replects required significant kernel changes, and the "replect kernel" did not have maintenance support after the graduation of its programmer,[1] following its completion.

In our measurements, the core structure and the access structure of R2D2 are pre-activated to eliminate the variance introduced by object activation, which may take many seconds. All active forms of R2D2 objects are concentrated on one node to reduce the number of remote invocations, which take longer than local invocations. A driver object (EFSMenu) sends the appropriate invocations to R2D2Root and measures the time it takes for R2D2 to service the invocations. For historical reasons[2], EFSMenu runs on a separate machine. Therefore, for read requests such as LookupSet, the active form of objects are concentrated on one node, except for EFSMenu (see figure 5.8).

The situation is more complicated for write requests such as ReplaceSet, AddSet, or DeleteSet, because the passive representations and PODs come into play. The passive representations reside on different file servers to increase availability, so checkpoint requests to different copies can proceed in parallel. Also for availability the active form of different copies of the same resource are activated on different nodes. Figure 5.9 shows the configuration of a write request to two copies.

[1] A. Proudfoot did his M.Sc. thesis work on replects [72].

[2] At the time of first measurements (early 1985), our SUNs had only 2 MegaBytes of main memory; EFSMenu caused paging and consequently timing variances if was run on the same machine.

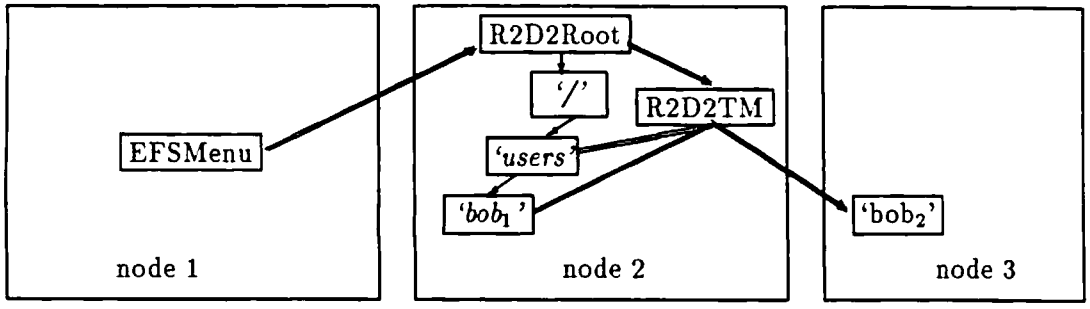

Figure 5.9: Writing Two Copies

5.3.3 Measurements

Using the set-up described above, we measure the elapsed time of requests by taking the timestamps before and after each invocation. For example, to measure the write request time, the code in EFSMenu looks like this:

```
Kernel.TSCurrent(tictime)
RepDir.AddSet(R2D2RootCapa, stringname, ... )
Kernel.TSCurrent(tactime)
```

Since the SUN internal timer advances in ticks of $\frac{1}{60}$ second, the granularity of timestamps is a problem. For relatively short invocations, we need to make a bundle of them to decrease rounding errors introduced by the clock. For example, a LookupSet invocation takes only a fraction of a second, so usually we repeat it 100 times before we stop the clock.

Typically, a measurement is repeated 10 times and the results averaged to obtain the confidence interval according to the formula derived from the Student t distribution:

$$Prob\left\{|\bar{x} - \mu| \leq \frac{S \cdot t_{\alpha/2}}{\sqrt{n}}\right\} = 1 - \alpha$$

where $\bar{x}$ = sample mean, s^2 = sample variance, μ = population mean, and n = number of samples. (In case $n = 10$ we have $t_{\alpha/2} \approx 2.8$ for $\alpha = 0.02$.) An example of actual numbers obtained in such a run appears in tables A.9, A.10, A.11, and A.12. These runs are obtained late in the night with no other activity in the network or nodes.

The performance of R2D2 is compared to the performance of a non-replicated directory in figure 5.2. In both cases, EFSMenu was used as the timing tool. The read operation was *Lookup('users/bob/mailbox')* and the write operation was *Replace('users/bob/test')*. The configuration of the non-replicated directory is analogous to the R2D2 configuration as described above. All objects in the directory hierarchy run on the same node, while their passive representations reside on a disk server.

		Non-replicated	R2D2	Comparison
Read	number of invocations	1 remote + 2 local	1 remote + 4 local	
Read	time	0.23 ± 0.01	0.42 ± 0.01	1.8 times
Write	number of invocations	1 remote + 2 local	1 remote + 4 local	
Write	number of checkpoints	1	2	
Write	time	1.43 ± 0.07	2.18 ± 0.08	1.5 times

The measured times show the confidence range at a confidence level of 98%.

Table 5.2: Measurement Summary (time in seconds)

5.3.4 Evaluation

For read invocations, the dominant cost factor is the invocation time. Examining tables 5.1 and 5.2, we see that the time taken by the non-replicated directory to service a Directory.*Lookup* can be attributed entirely to the invocations. Since the Regeneration method reads only one replica, we had hoped that the read overhead would be less. In fact, R2D2 read is 80% more expensive than non-replicated directory. The additional cost is due primarily to two extra invocations, introduced by the access structure (Root and R2D2TM), which also added some software overhead (about 10% of total).

The software overhead could be decreased by carefully recoding the R2D2TM. For example, the debugging and tracing code could be streamlined to save execution time. However, bypassing the extra invocations requires modification of R2D2's structure. As we have mentioned earlier in section 5.1.4, making the R2D2TM take care of concurrency and reliability atomicity introduced the two extra invocations. We could eliminate the two extra invocations by collapsing the R2D2Root, the R2D2TMs, and the top-level RepDirs into one object (of the replect kind). Since the purpose of R2D2 is to demonstrate the practicality of the Regeneration method, no further attempts were made to optimize R2D2 to circumvent this performance problem particular to Eden.

For updates, the most important cost factor is the checkpoint operation, which atomically transfers data to stable storage. Checkpoint operations take times at least an order of magnitude longer than those of invocations. Consequently, the time it takes to service any invocation which involves checkpoints is dominated by the number of checkpoints used. We speeded up R2D2 write operations considerably by making the R2D2TM send the RepDir update invocations (which cause checkpoints) in parallel. Compared to the non-replicated directory that checkpoints only once,

R2D2 takes only one and a half the time to checkpoint two copies.

With the performance measurements we have completed the work on data availability through replication. We have separated the data restoration from hardware repair. When data restoration can be done faster than hardware repair, Regeneration takes advantage of this separation to provide higher data availability than other replication methods. Given enough spare nodes and storage, a probabilistic analysis shows the advantage of Regeneration. The replicated directory system built with Regeneration, R2D2, will be used in the implementation of the nested transaction mechanism, described in chapter 6.

Chapter 6

Eden Resource Management System

6.1 Overview

Our nested transaction system is called Eden Resource Management System (ERMS). The main goal of ERMS is to investigate the design and implementation of a powerful transaction mechanism using composition. We use composition in two ways: statically, we compose modules to form elaborate objects; dynamically, we compose objects to form large structures. Eden objects are ideally suited for this purpose; since EPL supports modules and invocations, both static and dynamic composition are easy.

However, we must recognize the performance limitations of the Eden prototype (table 5.1). Since Eden object programmers must use a small set of expensive primitives, raw performance is not our primary goal. Rather, we favor clarity and generality in ERMS. Performance considerations are taken into account at the level of checkpoints, whose cost is in the order of seconds.

6.1.1 The Ideas

The key technique used in ERMS is *composition*. By composition we mean the combination of objects (and modules) to perform tasks that individual objects (and modules) are unable to accomplish. In addition, the components used in the combination should be modules – preferably existing software – implementing well-known algorithms.

The fundamental building block of ERMS is the ERMS Transaction Manager (ETM). Each ETM handles concurrency control and crash recovery of one transaction.[1] ERMS supports nested transactions in a tree hierarchy by organizing ETMs into trees that reflect the structure of nested transactions.

The key idea of ERMS is to carefully compose concurrency control and crash recovery information in a tree structure isomorphic to the hierarchy of nested transactions. One unique characteristic of ERMS is its adoption of concurrency control and crash recovery methods which are exactly the same as those used in single-level transaction systems. Although the data structures and protocols used at each level are the same as the single-level transaction systems, composition provides nested concurrency atomicity and reliability atomicity.

[1] To the best of our knowledge, the idea of one transaction manager per transaction was first suggested by Jessop [46].

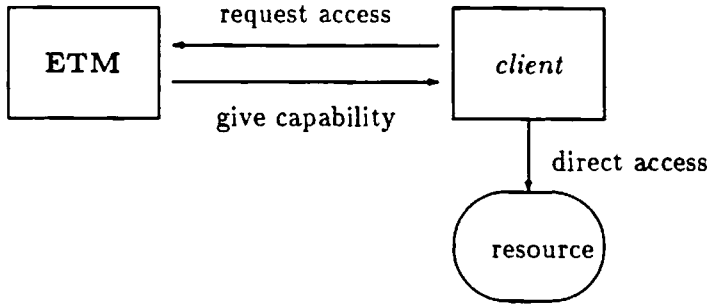

Figure 6.1: Client and ETM

Although various concurrency control methods could be used, we have chosen two-phase locking and version-based recovery. At the top level, a system lock manager synchronizes resource access and a system directory handles recovery. For each transaction, the main data structures are a lock table to manage locks and a mapping to keep track of committed versions. We concentrate these data structures in the ETM, which in two aspects differs from the transaction managers in the TM/DM model introduced by Bernstein and Goodman [9]. First, their DM maintains the data structures for concurrency control and crash recovery. The ETM contains both. Second, their transaction manager takes care of all transactions running on that node. In comparison, an ETM serves only one transaction. However, these differences are in implementation, and the ETM maintains Bernstein and Goodman's abstraction of transaction manager, relieving concurrency control and crash recovery from the clients. Using one ETM per transaction facilitates composition, so the design, implementation, and presentation of ERMS have been simplified considerably.

6.1.2 Computation Model

Figure 6.1 shows the high level model of resource access in ERMS. A client object invokes the ETM requesting access to resources (one resource is shown in the figure). After taking the necessary steps to assure concurrency atomicity and reliability atomicity, the ETM gives the client direct access to the resources, which are sets of Eden objects when replicated. Unlike the Argus user-defined atomic data types [83], our resource objects do not include any transaction concurrency control or crash recovery. The ETM is responsible for both concurrency atomicity and reliability atomicity of all resources in a transaction.

Since ETM is separate from the client, multiple clients may participate in the same transaction. The client who started the transaction may pass the ETM's capability to other client objects, or to other processes within the same object. Figure 6.2 contains an example showing multiple threads of control in a hypothetical transaction example.

In the distribution of client objects and inclusion of multiple processes within each object, ERMS differs from usual distributed transaction systems. In the Argus language [56] and R* [55] distributed database, a process corresponds to one transaction. In ERMS, many processes from many client objects may participate in one single transaction. Distributed client objects com-

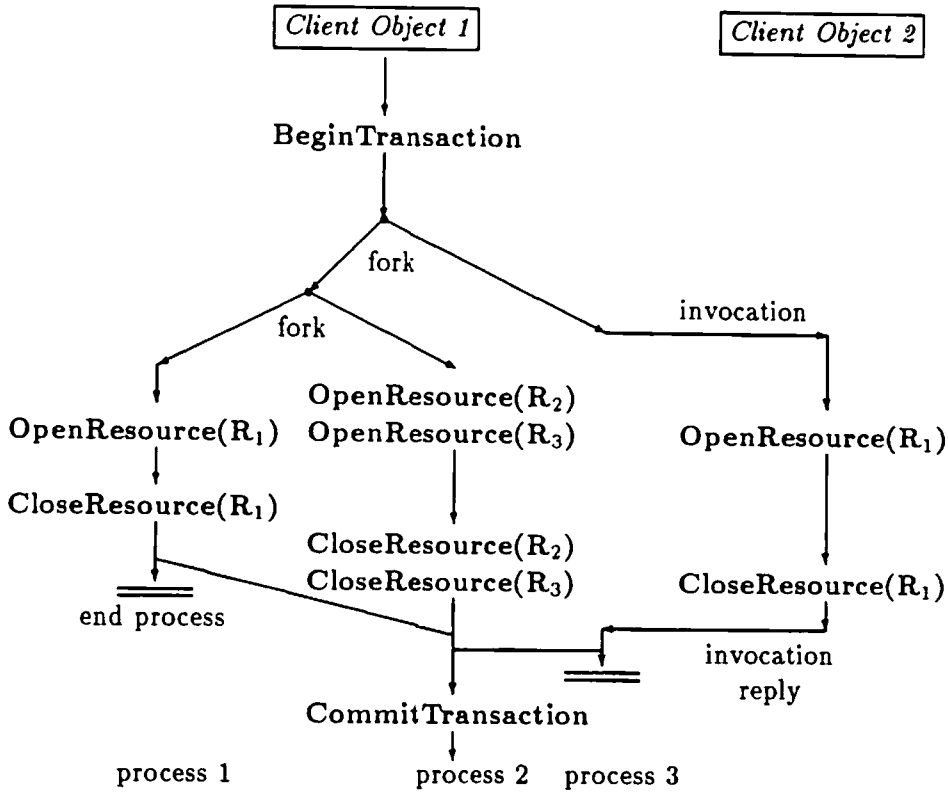

Figure 6.2: Hypothetical Transaction Example

municate with each other through Eden's location-transparent invocations. Although resources internal to an Eden object (an EPL program) may have their access synchronized by monitors, resources external to the object cannot. Since there are no inter-object synchronization mechanisms, resource accesses from client objects are controlled by the one ETM for the transaction. Therefore, the ETM conveniently encapsulates concurrency control and crash recovery.

6.1.3 Client Interface

The interface between ERMS and clients is explicit. At the beginning of the transaction, the client obtains an ETM that becomes responsible for the transaction, and tells the ETM that the transaction has started. Similarly, at the end of transaction, the ETM must be told explicitly whether the transaction has committed or aborted. Three **TransactionBracket** operations inform the ETM of the beginning and end of the transaction.

- BeginTransaction: beginning of transaction.
- AbortTransaction: end of failed transaction. Rollback all updates performed since BeginTransaction.
- CommitTransaction: end of successful transaction. Execute atomically all updates since BeginTransaction.

At BeginTransaction, it is assumed that all resources are in a consistent state. The client proceeds to make temporary changes to resources. When changes are completed, CommitTransaction atomically makes them permanent. If for some reason the client is unable or unwilling to make the planned changes, AbortTransaction returns the resources touched by the transaction to their original state at BeginTransaction.

Resource access during the transaction is also explicit. The client tells the ETM which, how, and when resources are being accessed. Two **ResourceManagement** operations inform the ETM about the beginning and the end of access to each resource.

- OpenResource: request access to a resource.
- CloseResource: return a resource after use.

At OpenResource, the ETM uses a concurrency control method to serialize resource access. If there are no conflicts, the capability of the resource is returned. The client reads and writes the opened resource directly with the capability. After use, CloseResource returns the resource to the ETM so other processes or clients within the transaction may access it. If the transaction ends (either by commit or abort) before an opened resource is returned with CloseResource, all updates on that resource are lost; the resource remains in its original state before the OpenResource.

This interface is uniform for top-level and subtransactions. The same syntax is used for both, and the client chooses which case applies at run-time. The main advantage of this uniformity is that it allows unrestricted composition of application transactions.

From the client point of view, OpenResource and CloseResource may seem redundant, since resource accesses invocations themselves mark the scope of access. The explicit interface has

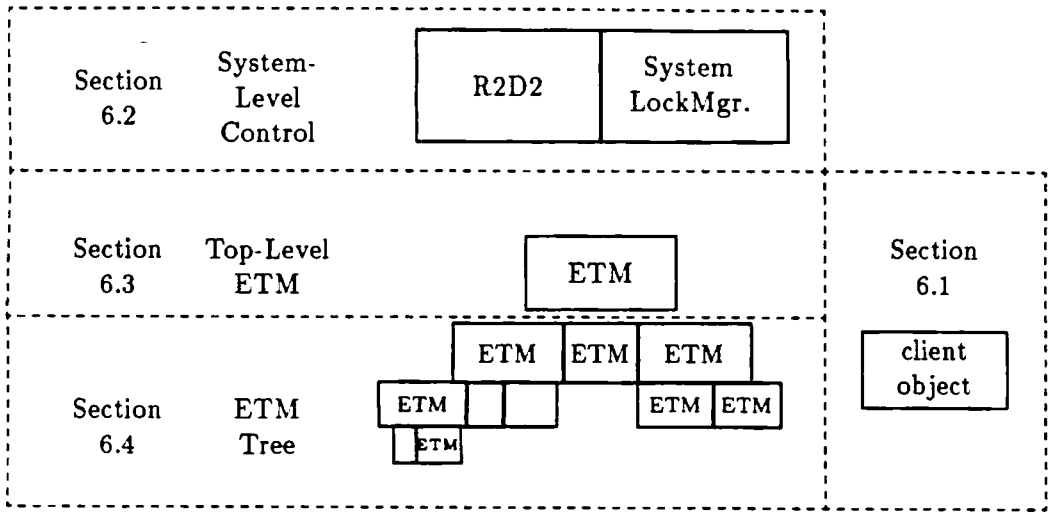

Figure 6.3: Schematic ERMS Structure

been dictated by the lack of compiler support in ERMS. Since the EPL compiler does not collect information on resource access invocations, ERMS is unable to implicitly open and close a resource at its first and last invocation, respectively. An effort to partially integrate ERMS and EPL resulted in a plan, which remains to be executed, however.

The same example in figure 6.2 (page 50) shows a possible execution sequence for a hypothetical transaction. Client object 1 starts the transaction, and forks two more concurrent processes. These processes permit the client to exploit the parallelism in a distributed system. For example, one of the processes invokes a second client object, bringing it into the transaction. The multiple accesses to resources R_1, R_2, and R_3 are all synchronized by ETM. All boldface operations shown in the figure are invocations to the transaction's ETM.

The schematic structure of ERMS is shown in figure 6.3. We have already introduced our computation model and client interface in sections 6.1.2 and 6.1.3. The highest level of concurrency control and crash recovery is the system-level control, described in section 6.2. Section 6.3 explains the top-level transaction manager, and section 6.4 describes the nesting of transaction managers to provide nested concurrency control and crash recovery. We summarize ERMS features in section 6.5 and give an example application in section 6.6. Finally , we compare ERMS with other systems in section 6.7.

6.2 System Level Control

Figure 6.4 shows the system level support for concurrency control and crash recovery of top-level transactions. At top left, a system directory (R2D2) stores the public version of resources, which may be replicated. At top right, a System Lock Manager (SLM) serializes resource access of top-level transactions. The double arrow indicates concurrency control links, and the single inclined arrow between R2D2 and ETM shows communications related to reliability atomicity. A top-level ETM is shown in the picture; the horizontal arrow represents client requests to the

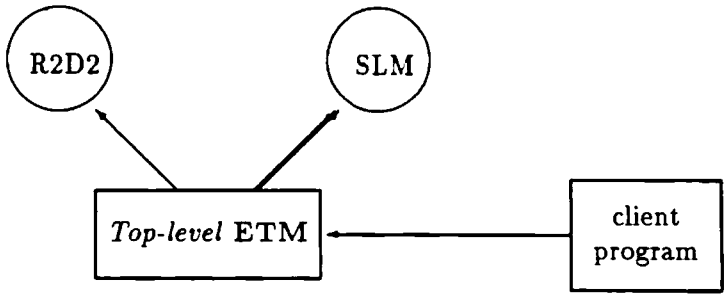

Figure 6.4: Top-Level Structure

ETM, such as BeginTransaction or OpenResource.

The building blocks used to construct the top level transactions are well-known single-level transaction techniques. We have chosen strict two-phase locking [30] and version-based recovery. In section 6.2.1, we describe the System Lock Manager, which controls the resource access at the top level. In section 6.2.2, we summarize R2D2's contribution to version-based recovery at the top level.

6.2.1 System Lock Manager

One Eden object, the System Lock Manager, maintains the lock table for all resources controlled by ERMS. The System Lock Manager object implements the abstract type **TwoPhaseLock**:[2]

- LockName(in: ResourceName, AccessMode, TID; out: Status) - If ResourceName is not locked, grant requested lock by inserting the triple (ResourceName, AccessMode, TID) into a lock table. Otherwise grant lock (if possible) according to lock compatibility table 2.1 (shared read, exclusive write).

- UnlockName(in: ResourceName, TID; out: Status) - Remove the entry that matches (ResourceName, TID) from the lock table.

TwoPhaseLock is also implemented by R2D2Root in R2D2 (section 5.2.2), and ERMS Transaction Manager (section 6.3). The same software module, **LockTable**, implements the abstract data type for System Lock Manager, ETM, and R2D2Root. LockTable is a mapping of resource names into the lock holders and their respective access modes (read or write).[3]

To avoid deadlocks, the System Lock Manager does not block a LockName request in case of conflicts. The invocation returns with a status indicating the type of the conflict (whether it is read-write, or write-write), and the TID (a unique Transaction Identifier) of the lock holder. Based on this information, the ETM making the request follows the wait-die deadlock avoidance scheme of Rosenkrantz et al. [75].

[2]The return status codes of TwoPhaseLock can be found in table A.4.

[3]More implementation details of LockTable's main data structure, LockMap, can be found in appendix section A.3.5.

The current System Lock Manager does not implement any time-out mechanisms for ERMS resources. Consequently, recovery from crashes usually requires some manual lock releases. There are two reasons for this omission. First, locking concurrency control was chosen because the LockTable module had already been implemented. Investigations into problems specifically related to locking are not part of our research goals. Second, time constraints on the building of an experimental system like ERMS prevented the exploration of side issues.

Following the strict two phase locking protocol, the ETM consults System Lock Manager the first time a resource is requested by an OpenResource. The ETM attempts to lock the resource name on behalf of the client. If there is a conflict, OpenResource fails and the resource remains inaccessible. If the lock is granted, System Lock Manager assures the serializability of resource access. Since we wish to enforce strict two phase locking, the lock is not released during CloseResource. Rather, the locks are released only when the transaction ends, either by commit or abort.

After appropriate actions making all changes permanent (commit) or reversed (abort), the ETM invokes System Lock Manager to release all locks held by the transaction. Each ETM has an internal LockTable module, which remembers the locks being held by this transaction. The ETM simply goes through the resource names in its own LockTable and releases those locks from System Lock Manager.

We should emphasize that all transactions in the above discussion refer to top-level transactions. The synchronization of subtransactions is controlled by parent ETM, to be explained in section 6.4. Consequently, the only ETMs that invoke System Lock Manager directly are those responsible for top-level transactions.

6.2.2 Version-Based Crash Recovery

The foundation of ERMS version-based recovery is R2D2, which stores the most recent committed version of resources. R2D2 is simply a directory, mapping resource names into sets of capabilities. The actual disk storage management is handled by Eden kernel (POD). The versions in R2D2 are "public", i.e., committed by top-level transactions.

Since we have already described R2D2 in detail (chapter 5), here we omit the description of its implementation. The most important fact is that R2D2 is a mapping – supporting invocations such as lookup, add, delete, and replace entries in the mapping. These invocations are summarized in appendix table A.1.

Version management in ERMS is similar to other version-based systems [48]. The first time a resource is opened, the ETM looks up the current version in R2D2. If the resource is opened for update, the ETM makes a copy of the resource, and passes the capability of the copy to the client. The client writes to the new version/copy directly.

At the time of CloseResource, the ETM cannot replace the version in R2D2 by the new one, since the transaction has not yet committed. The ETM must therefore save the new, temporary versions of resources, which have been closed in the transaction. In addition, ETM also remembers the appropriate action to be taken at the commit time. For example, a newly created resource must be added to R2D2, while an updated resource should be replaced. In any case, R2D2 will be involved with an opened resource only when the transaction commits.

The version mapping in ETM implements the abstract type RepDirectory, also supported by R2D2. The same software module, **RepDirTable**, implements the abstract type for both ETM in ERMS and RepDir in R2D2. More implementation details may be found in appendix section A.3.3.

If the transaction aborts, the responsible ETM simply discards the new versions created for that transaction. R2D2 remains untouched by the aborted transaction.

If the transaction decides to commit instead of abort, the new versions must replace the old versions in R2D2. The ETM makes a ReplaceSet invocation to update each resource being written with the new versions. It is only after the new versions have been put in R2D2 that the ETM releases the locks it has on the resources. Since all resources remain locked during the updates, all the updates of a transaction appear atomic to other transactions. Should the ETM be interrupted during the commit protocol, R2D2 would contain some of the new versions, and some of the old versions. Since the resources remain locked, the inconsistency is invisible to other transactions. This situation is described in more detail in section 6.3.

ERMS relies on the Eden garbage collector to delete the aborted new versions or replaced old versions. Had we been more interested in distributed garbage collection, we could have implemented some garbage collection scheme at the object level. In that case, R2D2 would have to contain a reference counter for each resource, and the ETMs would add and subtract from the counter as resources were opened and closed for access. When the counter reaches zero, ETM would invoke the object asking it to delete itself. With our concentration on the nesting of transactions, we left the object level garbage collection problem to future research.

6.2.3 Discussion

We should emphasize that there are no special features in either System Lock Manager or R2D2 that make them especially suitable for a nested transaction mechanism. Both System Lock Manager and R2D2 are standard components which could have been used in a single-level transaction mechanism. For example, Paxton [70] proposed a client-based transaction system, in which the underlying server provides three kinds of primitives: random access to files, file locks, and lock time-outs. R2D2 and System Lock Manager correspond directly to his file access and file lock services.

From the performance standpoint, we made a few necessary design choices which are suboptimal. First, the combination of locking concurrency control and version-based crash recovery does not take advantage of either. Locks permit write in-place, which usually is faster than creating a new version. However, write in-place requires a log-based recovery, which would either impose restrictions on the types of resources managed by ERMS, or make programmers provide type-specific recovery for each resource Edentype. In any case, the only Eden primitive to write to disk is checkpoint, which writes the object's entire state, so there is no kernel support for object-level logging. Not willing to expand Eden kernel capabilities, we chose version-based recovery instead of logging.

Once we are resigned to the performance penalties of a version-based system, we could adopt some other concurrency control mechanism which provides more concurrency. For example, timestamp methods may allow more transactions to run in parallel, accessing different versions of a

resource. We have used locking simply because the module LockTable had already been implemented, and it represented the shortest path to a working system. Other concurrency control methods based on timestamp intervals [6] have been implemented as part of another work [66], which rates experimentally effective concurrency allowed by different concurrency control methods. Another illustration that more concurrency can be obtained is the work by Bayer et al. [7] showing that two versions are sufficient for the support of multiple readers *and* an exclusive writer. With version-based recovery, there is no technical difficulty in introducing this level of concurrency in ERMS, but it was not essential for the main purposes of this study.

A third performance problem may have been introduced by R2D2, which is a distributed directory. A centralized directory would provide the same functionality, and require fewer invocations. This is of concern in Eden because invocations are relatively expensive. However, since ERMS supports replicated resources, the directory must be crash-resistant. We chose R2D2 since it is the only working crash-resistant directory system in Eden. The alternative, replects [72], is not in regular use. In the end, due to the number of checkpoints necessary to ensure transaction reliability atomicity, invocation costs in R2D2 were insignificant compared to total overhead.

Finally, we made an effort to circumvent an obvious bottleneck. Conceptually, a new ETM is created at the beginning of each transaction and destroyed at the end. However, because Eden objects are implemented as Unix processes, their creation takes several seconds. For efficiency, we reuse the ETMs whose transactions have ended. Since the System Lock Manager is a convenient central resource, it also manages the creation and reuse of ETMs. In practice, ERMS clients ask the System Lock Manager for an idle ETM, instead of creating a new one. If the allocated ETM is already active from the last transaction, the recycling saves up to several seconds per transaction. This function of System Lock Manager is unrelated to its role in concurrency control or the nested transaction mechanism. Some practical difficulties are introduced by this economic measure and their solution will be described in section 6.3.4.

6.3 Top-Level ETM

The main building block in ERMS is the ETM transaction manager. An ETM serves one transaction, which can be either a top-level transaction or a subtransaction. In this section, we describe the ETM serving a top-level transaction without subtransactions. ETM's control of a subtransaction and the composition of ETMs to provide nested transactions will be discussed in section 6.4. For both top-level ETMs and subtransaction ETMs, the data structures (in section 6.3.1) are the same. Moreover, the algorithms in sections 6.3.2 and 6.3.3 are independent of nesting aspects; therefore, they are also the same for both.

6.3.1 ETM's Data Structures

There are three main data structures in ETM: the LockMap, DirMap, and ChildList. ChildList maintains the list of capabilities of the ETMs serving the child transactions, and will be discussed in section 6.4. The LockMap and DirMap are used for concurrency control and crash recovery, respectively.

LockMap is part of the software module LockTable, which has been described in section 6.2.1.

The LockTable module implements the abstract type TwoPhaseLock, with the LockName and UnlockName operations. Although both a top-level ETM and System Lock Manager employ a LockTable, the function of LockTable in each is different from the other. When a resource is being locked in the System Lock Manager, an entry is placed in LockTable to signal the placement of a lock on that resource. In ETM, the same entry is placed in LockTable to remind the ETM that it has the lock on the resource, so the lock will be released at transaction completion.

As a simple reminder for the ETM, LockTable may appear an overkill. However, as we shall see in the next section, the top-level ETM also functions as the parent ETM of its subtransactions. Moreover, even at the top-level, several client processes may try to open the same resource at the same time. So we need the full power of LockTable to synchronize the resource access from concurrent processes and subtransactions.

DirMap is a mapping of resource names into sets of capabilities. (If this sounds familiar, it is because DirMap is also used in the Core Structure of R2D2.) The main function of DirMap is to remember the most recently closed version of a resource being updated. Since R2D2 stores only the committed versions, all intermediate versions produced in the transaction must be saved in DirMap before the transaction commits. In addition to the capability of the most recently closed version, DirMap also remembers the appropriate action to be taken at commit time. For example, a newly created resource must be added to R2D2, while an updated resource should be replaced. In section 6.3.3, we will see how atomic commit is accomplished in ERMS using DirMap. Appendix section A.3.3 contains more implementation details of DirMap.

6.3.2 Resource Management

In section 6.2.1, we have explained how the System Lock Manager supports TwoPhaseLock, and how the R2D2 provides a mapping in section 6.2.2. ETM's internal data structures have been described in section 6.3.1. Now we can put everything together, and using these components, show how client requests for resource access are made atomic

Resource access is bounded by the operations OpenResource and CloseResource, both specified in appendix section A.1.6. We describe the algorithms informally here; the operations are explained more concretely by the example in section 6.6.

OpenResource takes as **in** parameters the resource name and access mode (either read or write); it returns the capability of the appropriate version for client access. The most important part of OpenResource is the careful checking of resource access according to the lock compatibility (table 2.1) to assure serializable access. The algorithm for OpenResource is:

1. Check LockTable for resource name. There are three cases:

   (a) **First Time:** Resource name is not in LockTable.

   (b) **Already Open:** Resource name is in LockTable, but somebody (another process) has already opened the resource. In this case, the access is decided by lock compatibility table 2.1.

   (c) **Second Time:** Resource name is in LockTable, and nobody holds a lock on it, then the resource has been opened and closed.

2. If case 1a, the **First Time**, we need to request the appropriate lock from System Lock Manager. If successful, then insert the resource into LockTable and proceed to step 4. Otherwise, OpenResource fails (exit).

3. In case 1b, the resource is **Already Open**. If lock request is compatible with the current access mode – for example, read and read – proceed to step 5. Otherwise, OpenResource conflicts and fails (exit).

4. In case 1c, the **Second Time**, the ETM already holds a lock from System Lock Manager. If the LockTable indicates a read lock, and the Open request requires a write lock, then go back to step 2 to get the right lock. Otherwise, insert the lock holder into LockTable.

5. Check DirMap for resource name. If the resource name is not in DirMap, lookup the name from R2D2, insert the name and the most recently committed version into DirMap.

6. If resource is being opened for read, choose one capability from the set and returns it; increment reader count by one. Otherwise, make a copy and return the capability of the copy (exit).

CloseResource takes as **in** parameters the resource name and the new version's capability; it returns a status code. The algorithm for CloseResource is as follows:

1. If the resource has been opened for read, decrement reader count by one.

2. If the resource has been opened for write, from the new version, generate the right number of replicas at the right nodes, and insert the new version into DirMap. Remember resource status (new or replacement).

3. If reader count = 0, mark the resource as unlocked in LockTable.

We should observe that CloseResource does not touch R2D2 or System Lock Manager. It does not move the new version into R2D2 because the transaction has not committed. The lock is not released in System Lock Manager because of the strict two phase locking protocol, which releases all locks after the commit.

6.3.3 TransactionBracket

There are three operations in the abstract type TransactionBracket: BeginTransaction, AbortTransaction, and CommitTransaction. We now describe the actions taken by the ETM at each of the three.

First, BeginTransaction tells the ETM whether it is managing a top-level transaction or a subtransaction. The parameter that points to the parent capability is null in case of a top-level transaction. If the ETM is managing a top-level transaction, it initializes some variables, and returns control to the client.

Second, AbortTransaction ends the transaction, requesting the ETM to revert all resource changes, bringing them back to their original state. Since we have a version-based recovery, which makes the clients write on new copies, the committed versions always remain undisturbed.

Consequently, there is very little work to be done at abort. First, the ETM checkpoints the decision to abort. (Otherwise, it may crash and "forget" the abort at reactivation.) Then, it goes through the list of locked resources in LockMap and releases the locks from System Lock Manager. All temporary versions in DirMap are simply dropped and will be garbage collected by the Eden garbage collector.

Third, CommitTransaction is more delicate than abort for one reason: all the changes must appear atomic to the outside world. Our algorithm is very similar to the one described by Lampson and Sturgis [51]. The main idea is to re-execute the idempotent switching operations as many times as necessary until all new versions have replaced the old ones. Since this is the top-level ETM, the committed versions are installed in R2D2. The algorithm for CommitTransaction is:

1. Checkpoint the ETM: write the commit record, and DirMap writes all new versions to be committed to disk.

2. For each updated resource in DirMap, invoke R2D2 to replace the old version by the new version.

3. Once all new versions have been installed in R2D2, release all lock held in LockMap.

4. Checkpoint the successful end of commit.

Since we do not release the locks before all new versions have been installed, the updates in the transactions appear atomic to the outside world. The only remaining problem is to make sure a crash in the middle of the commit protocol will not make R2D2 inconsistent. Because the ETM checkpoints at the beginning of commit, its passive representation contains the state at step 1. The next time ETM is invoked, (for example, by the client that inquired about the outcome of commit) the Eden automatic reactivation will restart the protocol at step 2. Since R2D2 operations are idempotent, there is no harm in re-executing the initial replace operations again. Once step 3 has been reached, all versions have been installed and R2D2 has become consistent.

6.3.4 Discussion

The design choice of one ETM per transaction may have introduced a performance problem in ERMS. The current implementation uses an Eden object for an ETM, and since Eden objects are Unix processes, we pay a high performance penalty. In section 7.3, we shall see that this problem is not inherent to the design, and could be removed. The current situation is somewhat alleviated by the reuse of ETMs, which we mentioned in section 6.2.3. However, since an ETM is conceptually unique for its transaction, we have to protect this property of ETM carefully.

First, we have to make sure that the ETM does not confuse the clients of a previous transaction with the clients from the next transaction. Currently, each BeginTransaction returns a unique TID for the transaction, and clients of that transaction must provide the TID in every request to ETM (e.g. OpenResource). The ETM simply returns an error if the client's TID does not match the executing transaction.

Second, it may be useful for an ETM to remember the outcome of past transactions. For example, an orphan transaction may inquire about its parent's status. The current implementation simply returns an error, since the transaction outcome is always stored in the parent ETM, and committed transactions have obtained all "interesting" results. If an ETM ever receives the reply "I am not transaction TID" from its parent, then the child assumes that its own outcome is immaterial, and the ETM aborts immediately. If the parent receives this reply from a child, the parent concludes that the child has aborted and proceeds accordingly.

6.4 Nesting ETMs

In the previous sections, we have described the components of ERMS, and how they support the top-level transactions. Now we are ready to compose ETMs into a tree to provide nested concurrency and reliability atomicity. In section 6.4.1, we describe the maintenance of the ETM tree. In section 6.4.2, the nested concurrency atomicity is explained. In section 6.4.3, the nested reliability atomicity is described. Finally, in section 6.4.4, we summarize the structure of ETM.

6.4.1 ETM Tree

As we have suggested in figure 6.3, ETMs form a tree to provide nested transactions. The ETM tree is isomorphic to the tree-structure of nested transactions which the ETMs control. From the entire system point of view, all ETMs form a forest, where each top-level ETM is the root of a tree. The ETMs at the lower levels manage subtransactions.

The maintenance of the ETM tree is done by the participant ETMs. Each ETM keeps a list of capabilities, called ChildList, which contains the capabilities of the ETMs managing the subtransactions. For example, in figure 6.5, the capabilities of *sub*-TM_1 and *sub*-TM_2 will be in *Top-level*-TM's ChildList.

Each ETM's ChildList is filled by its child transactions. At BeginTransaction, the child ETM receives the capability of its parent. Immediately, the child ETM invokes its parent ETM, inserting its own capability into the parent's ChildList. Once on the ChildList, a capability remains there until the end of transaction. Therefore, the ETM Tree is dynamically constructed to reflect the current history of the top-level transaction and all subtransactions that ever started.

Each ETM in the tree has the capability of its parent ETM, and the capabilities of its child ETMs. For transactions with nesting level deeper than two, the ETM tree is distributed. In the following sections, we shall see that the ETM communicates only with its parent and children, so the distribution of ETM tree information eliminates unnecessary redundancy.

6.4.2 Nested Concurrency Control

In section 6.3.1, we have described the LockMap and summarized the LockTable module implementing the abstract type TwoPhaseLock. Now we proceed to explain the nested concurrency control by composing LockTables in the ETM tree.

First we state explicitly what we mean by nested concurrency control. By the definition of nested transactions, all subtransactions at the same level appear atomic to each other, and to their parent. We interpret this rule the following way. At the same level or above, invocations

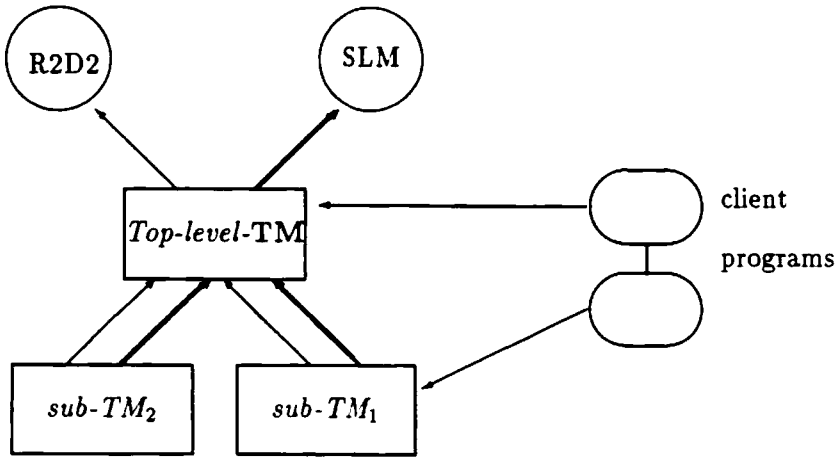

Figure 6.5: ETM Tree Example

enclosed in a transaction must happen atomically, as one operation. In other words, all partial results in the subtransactions must be hidden. In contrast, from below (a subtransaction point of view), all partial results produced by the higher level transactions must be visible.

The main purpose of this section is to demonstrate that we do not need special locking rules to implement these nested visibility scopes. We show that the composition of ETMs (and their LockTables) in a tree isomorphic to the tree structure of nested transactions is sufficient to hide partial results from subtransactions, and make visible partial results from higher transactions.

To facilitate the explanation, we introduce the analogy of "rent". When a resource is locked, we consider the resource to be "rented" to the lock holder. Therefore, at the top level, an ETM rents a resource from System Lock Manager the first time the resource is opened. Objects outside the transaction are locked out by the System Lock Manager, which has already rented the resource to the ETM. The ETM only sublets its resources to clients holding its capability, synchronizing their access through its own lock table. Consequently, as long as the resource has been closed, the ETM sublets the resource to another client process each time the resource is opened.

The key observation is that a child ETM may also open a resource and rent it from its parent, as if the child ETM were a client. There is no difference from the parent ETM point of view, since it simply sublets the resource according to the lock compatibility table. However, the child ETM now may sublet the resource to its own clients, and retains the resource when its own client closes it. The subtransaction's ETM will release the resource in the parent only when it ends, either by committing or aborting. Consequently, the clients of the parent transaction will be unable to rent the resource while the child ETM is running. Therefore, the intermediate results produced by the child transaction are hidden from the parent and sibling transactions.

The algorithms for OpenResource and CloseResource in a subtransaction ETM are exactly the same as the ones for a top-level ETM, described in section 6.3.2. The only change is that now the ETM invokes its parent ETM -instead of the System Lock Manager- to obtain the lock.

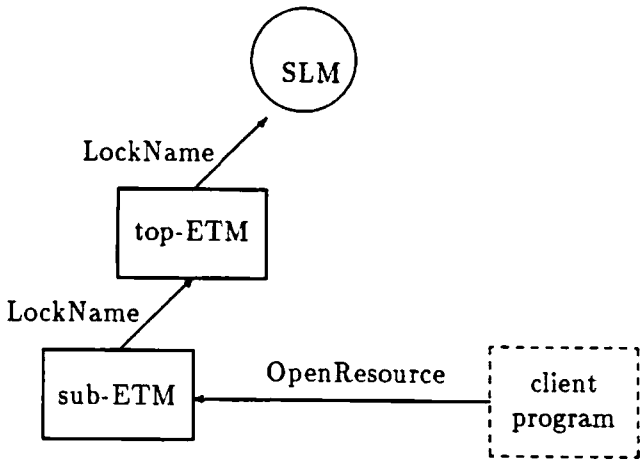

Figure 6.6: OpenResource: Nested Locking

Since the parent cannot sublet a resource without renting it from the grandparent, the chaining assures serializable resource access at each level. Figure 6.6 shows the lock requests caused by an OpenResource in a subtransaction when the resource is being opened for the first time.

In the general case, the cascading of lock requests stops at the lowest ancestor ETM that has opened the resource once. This ETM has rented the resource from its parent during the open, so it makes the decision on whether to sublet the resource. In case the resource is opened for the first time (figure 6.6), the System Lock Manager makes the decision. The rules for conflict resolution can be summarized as follows:

1. The parent resolves the conflicts between its children.

2. The lowest common ancestor resolves conflicts between branches in the ETM tree.

3. The System Lock Manager resolves the conflicts between top-level transactions.

The lowest common ancestor is the arbiter because of the cascading of lock requests: the two different branches are represented by two children of the lowest common ancestor, roots of the branches, and case two reduces to case one.

The LockMap has some static properties. The first is that each LockMap is a subset of the parent ETM's LockMap. This happens because every lock granted by the LockMap has to be obtained beforehand from the parent. Second, because we have adopted strict two-phase locking, the LockMap grows monotonically during the transaction until the commit point. At the end of a subtransaction, either commit or abort, all locks are released only in the parent's LockMap, not higher in the hierarchy. Third, since we use the wound-wait deadlock avoidance scheme, there is no need for deadlock detection or resolution.

6.4.3 Nested Crash Recovery

Nested crash recovery is analogous to nested concurrency control. By definition, subtransactions must appear atomic to their siblings and parent. In case a subtransaction aborts, we have to undo its effects. If a subtransaction commits, we have to make the commit appear atomic to its siblings and parent.

The main purpose of this section is to show that we do not need special mechanisms to implement the nested commit and abort. We show that the composition of ETMs (and their RepDirTables) in a tree isomorphic to the tree structure of nested transactions is sufficient to commit results atomically, and to undo all effects from aborted transactions.

To facilitate explanation, we introduce the analogy of caching. The top-level RepDirTable can be seen as a cache for R2D2; a child ETM's RepDirTable is a cache for the parent's RepDirTable. There are two techniques to handle updates in a cache: *write-through* (or store-through) propagates the changes immediately, while *write-back* (or copy-back) makes all the changes at a later time. The updates in RepDirTable do not write-through to the parent or to R2D2 because the transaction may abort, but are retained for possible commitment. If the transaction aborts, its ETM including RepDirTable is simply discarded. If the transaction commits, its RepDirTable executes write-back to its parent's RepDirTable (or at top-level, R2D2), which retains the changes until its own commit.

The operations on RepDirTable are thus operations on a cache. If the resource name is not found in RepDirTable, it asks the parent for the pair (resource name, set of capabilities). Then the operation is performed. For a cache with only one update operation –write– one "dirty bit" is sufficient to signal the need for write-back. However, we have three update operations –add, replace, delete– so each resource has a small finite automaton ("dirty state") to remember the appropriate update action to be carried out at transaction commit.

OpenResource is exactly the same as the algorithms described in section 6.3.2. The only difference is that the subtransaction ETM consults its parent ETM instead of R2D2. The parent's RepDirTable contains the most recent version for the subtransaction, since the operations on resources within the higher level transactions are visible to the subtransaction. For example, the results of an earlier committed subtransaction appear in the parent's RepDirTable, but nowhere else. If the parent ETM does not have the resource in its RepDirTable, then the resource is being opened for the first time in the parent transaction. In this case, the parent ETM consults the grandparent, and so on. Figure 6.7 shows the first time in the entire top-level transaction, when the cascading lookup requests end at R2D2. In general, the chain of lookup requests stops at the first ETM that has opened that resource once.

CloseResource does not involve communications with the parent, so the algorithm described in section 6.3.2 applies directly.

The protocol for a subtransaction commit is exactly the same as the one for a top-level transaction commit, described in section 6.3.3. The only difference is that the subtransaction ETM replaces the committed versions in its parent ETM, instead of R2D2. Putting the new versions in the parent ETM makes them visible to other sibling subtransactions and client processes in the parent transaction. At the same time, the parent retains control over the new versions, and so, should the parent transaction decide to abort, these versions will not be seen outside. Figure 6.8

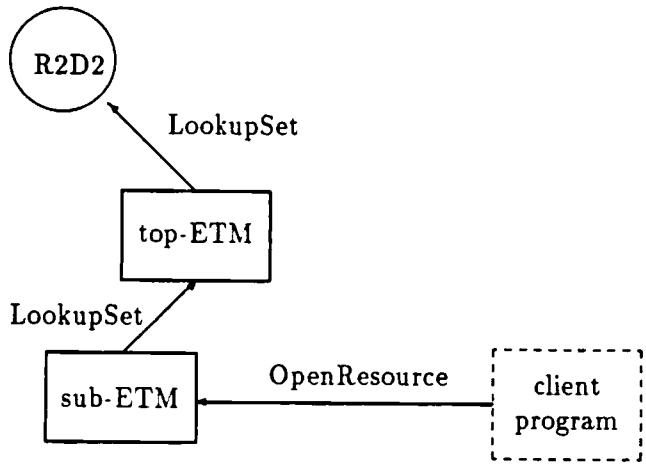

Figure 6.7: OpenResource: Nested Lookup

shows the commit actions limited to the parent ETM.

The protocol for aborting a subtransaction is more elaborate. A subtransaction ETM does four things when it is told by its client to AbortTransaction. First, it checkpoints the decision to abort the transaction. Second, the ETM informs its parent ETM of the abort. This is not strictly necessary, since the parent would inquire the status of all children at its own commit time. However, requiring the parent to ask for the children's statuses may be time consuming, especially when some children may have to ask their descendants. Thus, we choose to inform the parent immediately on the subtransaction outcome.

The third thing is to go through the ETM's own ChildList, telling each child ETM to abort. If a child transaction has already committed, it remains committed, but as we have seen, its results will be discarded anyway. A transaction may abort because of crashes, either of its clients or of its ETM. If an ETM crashes before it can tell its child ETMs to abort, the subtransactions will continue to run. These run-away subtransactions are called *orphans*. As soon as the parent ETM is reactivated, for example, by the completion of an orphan, all orphans are terminated by the parent.

Having checkpointed the decision to abort, informed its parent of that decision, and aborted all its children, the ETM proceeds to unlock the resources being locked by the transaction. At the top level, the ETM unlocks the resources in System Lock Manager. As a subtransaction, the ETM releases the locks it is holding from the parent ETM, which has 'sublet' the resource to the child ETM.

The DirMap in RepDirTable has some static properties similar to the LockMap. The first is that the resource names in the DirMap are a subset of names in the parent ETM's DirMap, since all resources are "rented" from the parent. However, the sets of capabilities, representing versions, may differ from ETM to ETM, since each transaction may write to resources independently. The second property is that DirMap grows monotonically through a transaction. The reason is that

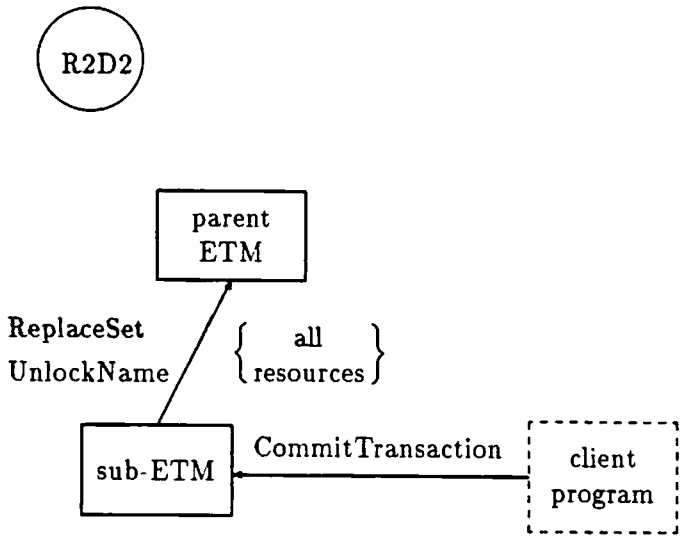

Figure 6.8: Subtransaction Commit

DirMap contains the history of updates to the resources, so each entry is used at the transaction commit to decide what action should be taken with respect to the parent's DirMap (add, replace, or delete).

6.4.4 Summary: Structure and Interactions

Summarizing the discussion of previous sections, the ETM is composed of four modules: **RepDirTable**, **LockTable**, **TreeManager**, and **ResourceManager**. RepDirTable maintains the most recent version of the resources being accessed. LockTable keeps track of locks on these resources. TreeManager takes the appropriate actions at the beginning and end of the transaction. ResourceManager implements the OpenResource and CloseResource protocols, hiding crash recovery (RepDirTable) and concurrency control (LockTable) from clients.

There are some static properties linking a parent ETM's modules and its children's (see figure 6.9). First, an ETM's DirMap is a subset of its parent ETM's DirMap. Every resource that a child controls has its previous version in the parent's DirMap. Similarly, an ETM's LockMap is a subset of its parent ETM's LockMap. Moreover, the locks held by the parent must be equal to or stronger than those of its children. For example, if a child obtains a write lock on a resource, its parent must hold a write lock on that resource given by the grandparent. Finally, each parent's TreeManager keeps a list of its children's capabilities. These lists form the transaction tree.

There are also rules governing the dynamic interactions between a child ETM and its parent. First, a child knows only its immediate parent, there being no direct communication between a child ETM and its grandparent; this rule applies to all four modules. Second, if the RepDirTable and LockTable need information from the parent to service some requests, they make invocations that are serviced by their parent's RepDirTable and LockTable. Consequently, a

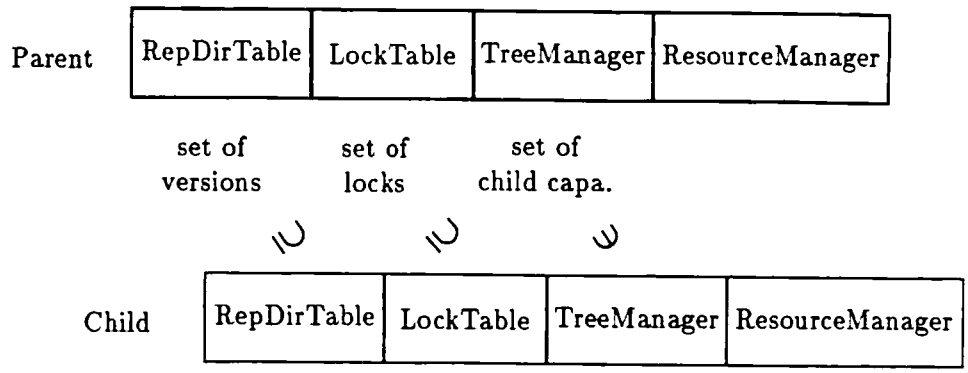

Figure 6.9: Relationships between the Parent and Child ETMs

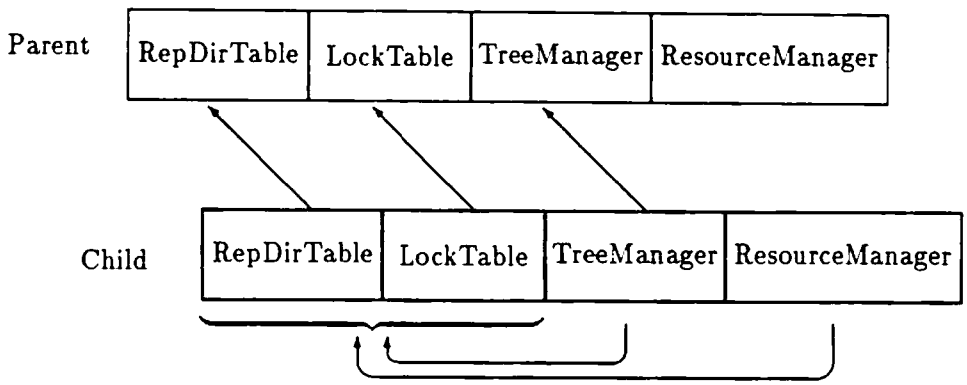

Figure 6.10: Communications between Parent and Child ETMs

ResourceManager needs only call the local RepDirTable and LockTable to synchronize resource access. Similarly, a TreeManager communicates with the local RepDirTable and LockTable, plus its parent's TreeManager. The communication paths are shown in figure 6.10.

6.5 Summary of ERMS Features

The strength of ERMS is in the economy of design concepts and the generality of implemented features. All ERMS components described in the previous sections use well-known techniques for single-level transactions. Careful composition of these techniques in the ETM tree provides the key for nesting in ERMS. Not only does the composition facilitate the concrete design and implementation of ERMS, but also the composition allows other combinations to form new designs of nested transactions, discussed in chapter 7.

ERMS features may be divided into two groups: resource support and transaction support. We summarize resource support in section 6.5.1, and transaction support in section 6.5.2.

6.5.1 ERMS Resource Support

Resources controlled by ERMS have three characteristics:

1. Type generality.
2. Location transparency.
3. Replica transparency.

First, ERMS resources may be objects of any type. In Eden, this is a necessity rather than option. As we have mentioned in section 5.1.2, Eden objects can service any invocations that the Edentype programmer has programmed in. Even if we take into account all current invocations, new ones may be invented any time. Consequently, ERMS cannot restrict operations supported by its resources.

Although ERMS controls resource access for any Edentype, it does distinguish 'read' operations from 'write' operations. Invocations that do not alter resource state are 'read', while invocations that do, are 'write'. At OpenResource clients must tell ERMS how they want to access the resource, so appropriate recovery actions may be taken.

In a way similar to Argus user-defined atomic data types [83], some resources may have their own concurrency control and recovery mechanisms. These resources may be accessed outside of ERMS control (bypassing OpenResource and CloseResource), since the use of ERMS is explicit and voluntary. Currently, there are no such resource types in Eden.

Second, all ERMS resources are Eden objects. Since Eden objects are location-independent, once in possession of their capabilities a client can access them regardless of their location in the network. Consequently, ERMS maintains resource location independence in Eden.

Third, ERMS resources are known by a string name, which is mapped into a set of capabilities. The resource access interface is the same regardless of the number of copies. During CloseResource of an update, the ETM generates the right number of copies, co-located with the original copies. The number of copies of a resource is stored in R2D2, and it may be changed at any time.

6.5.2 ERMS Transaction Support

ERMS transaction support has four components:

1. nested concurrency atomicity,
2. nested reliability atomicity,
3. top/sub syntax transparency,
4. long-term transactions.

First, in regard to its siblings and parent, each subtransaction is atomic. Nest concurrency atomicity means that intermediate results of a transaction T are invisible to its siblings and parent. However, T's subtransactions are parts of client processes, and they may read and use T's intermediate results.

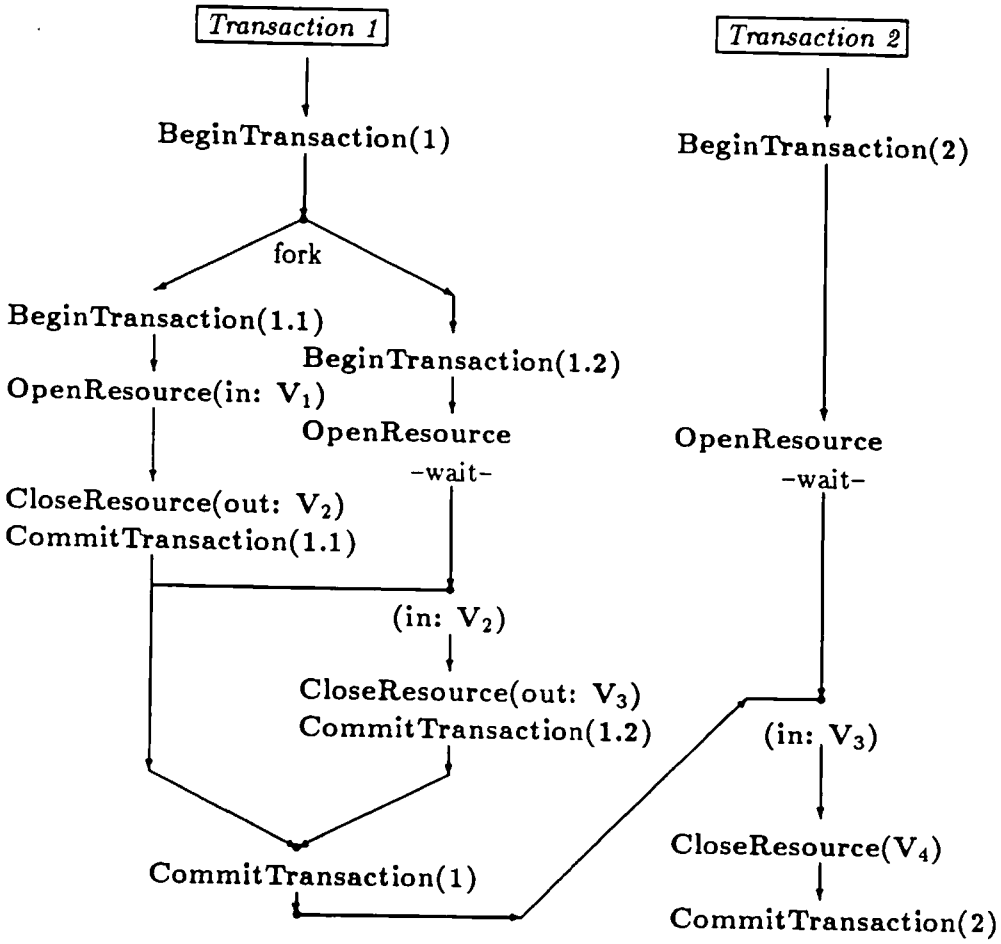

Figure 6.11: Nested Concurrency Control Example

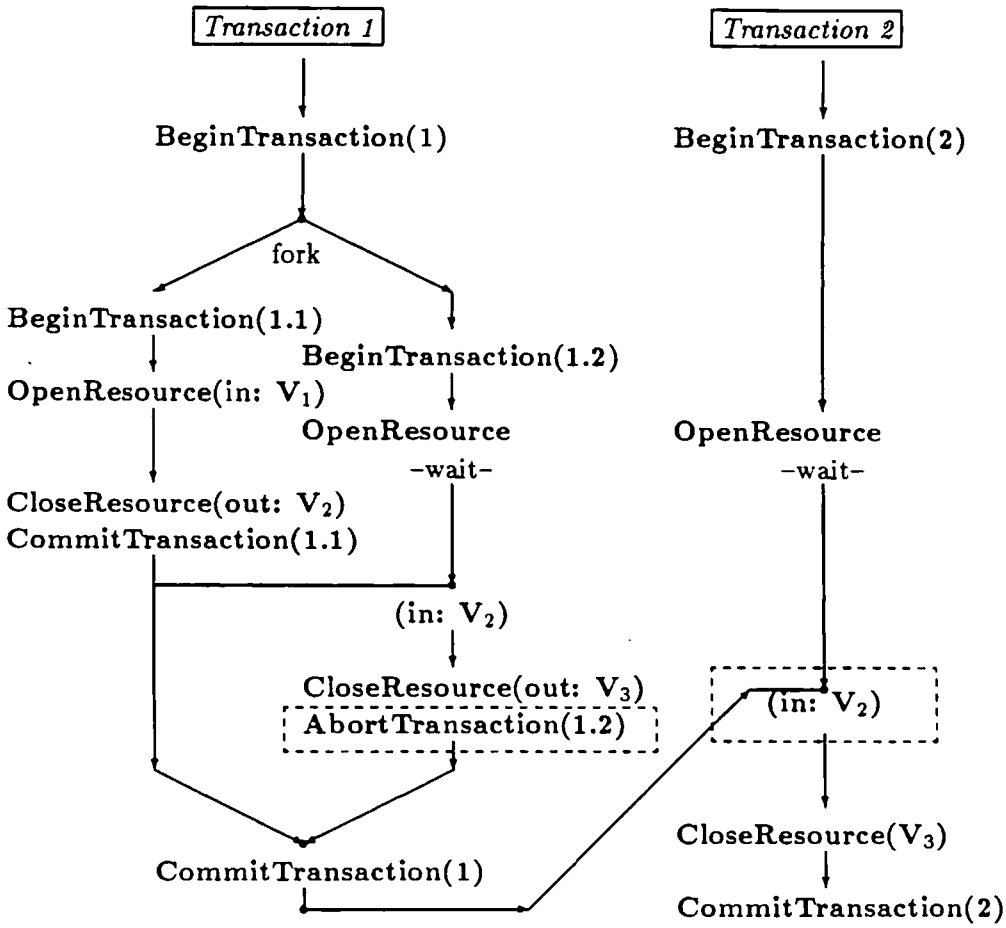

Figure 6.12: Nested Reliability Atomicity Example

In figure 6.11, a hypothetical example illustrates the conflict resolution at different levels. In the example, both Transaction 1 (T_1) and Transaction 2 (T_2) open the same resource (name omitted) for update. T_1 runs two subtransactions, $T_{1.1}$ and $T_{1.2}$, and both update the same resource. In figure 6.11, $T_{1.1}$ opens the resource first, and proceeds to generate a new version (V_2). Now both $T_{1.2}$ and T_2 open the resource, and they have to wait for the exclusive lock to be released. As soon as $T_{1.1}$ commits, its result (V_2) becomes available to $T_{1.2}$; a horizontal line shows $T_{1.2}$ receiving V_2 and generating V_3 as output. However, T_2 has to wait until the end of T_1 to proceed. In this case it receives V_3 and produces V_4.

The second component is nested reliability atomicity:

- Each subtransaction may abort independently of each other and the parent.
- Each aborted transaction has all the updates within it undone, including the results from some committed subtransactions.

In the following figure 6.12, we have the same example as figure 6.11, but with subtransaction $T_{1.2}$ aborted instead of committed. As shown in the same figure, the enclosing transaction T_1

can still commit. Since the subtransaction has aborted, the version it produced is dropped, and the result from $T_{1.1}$ prevails. To emphasize the differences from figure 6.11 to 6.12, the new parts are enclosed in dashed boxes. However, if T_1 aborts, the results from the subtransactions do not matter, and T_2 receives the original version V_1 of the resource.

Third, the syntax for top-level transactions and subtransactions is the same. Since the ETM is initialized at run-time, the same client program can choose to run as a top-level transaction or subtransaction, simply by sending the appropriate parameter at BeginTransaction. As we shall see in chapter 7, this dynamic selection brings new possibilities.

Fourth, long-term transactions were easily added to ERMS. During the first implementation of ETM, we realized that making the ETM "crash-proof" is sufficient for the restart of long-term transactions after a crash or planned deactivation. Since Eden checkpoint does the job, we simply introduced optional checkpoints at critical state transitions in ETM (mainly during OpenResource and CloseResource). The ERMS solution for long-term transactions is extremely simple. However, long-term transactions are more expensive than normal transactions since both the client and the ETM will have to checkpoint more often.

With the current strict two-phase locking policy, allowing multiple readers or an exclusive writer, ERMS long-term transactions may restrict resource access. Resources being updated in a long-term transaction remain locked for the duration of the transaction, decreasing potential concurrency in the system. The usef of timestamps, mentioned in section 6.2, would allow more transactions to execute in parallel. Some works [5] related to long-term transactions indicate the need for non-serializable operations. More research remains to be done on the application of long-term transactions to determine the necessary ingredients to make them more useful.

6.6 Application Example: Smart Bank Machine

To demonstrate the use of ERMS, we have written a program which mimics a bank machine. The demonstration system consists of two Edentypes: Bankomat and BankAccount . Bankomat is the client that uses ERMS to access BankAccount resources. A BankAccount object is basically an integer representing a certain amount of money, and the procedures to operate on the integer.

Each customer of the fictitious "EdenBank" has three accounts, checking, savings, and Visa, which are all instances of the Edentype BankAccount, but are distinguished by their resource name (for example, "customer/checking"). The operations a customer can do are:

- Withdraw "money" from any account.
- Deposite "money" to any account.
- Transfer "money" between any accounts.
- Print the balance of any account.

6.6.1 Sample Session

In figure 6.13, we show a sample session with the Bankomat, transcribed from the terminal. The sequence of activities generated by the conversation is illustrated in the schematic figure 6.14. The

```
(1)   Please enter your account name> calton        { begin top-level}
      w:withdraw, d:deposit, t:transfer,
      p:print, h:help, c:commit, a:abort
(2)   EdenBank> t
      Transfer from: c=checking, s=savings, v=Visa
      EdenBank>> c
      Transfer to: c=checking, s=savings, v=Visa
      EdenBank>> v
            Transfer amount = >$ 100
      w:withdraw, d:deposit, t:transfer,
      p:print, h:help, c:commit, a:abort
(3)   EdenBank> p
      Print balance of: c=checking, s=savings, v=Visa, a=all
      EdenBank>> a
            Balance for account EdenBank/calton
                  checking    savings     Visa
                  $ 40        $ 500       $ 100
      w:withdraw, d:deposit, t:transfer,
      p:print, h:help, c:commit, a:abort
(4)   EdenBank> w
      From: c=checking, s=savings, v=Visa
      EdenBank>> c
            Withdraw amount (integer) = >$ 100
      w:withdraw, d:deposit, t:transfer,
      p:print, h:help, c:commit, a:abort
(5)   EdenBank> c                                   { end top-level }
```

Figure 6.13: Sample Bankomat Session

whole session is a top-level transaction, composed by operations enclosed in subtransactions. At the first glance, there is no need for subtransactions, since the operations are sequential. However, EdenBank allows simultaneous sessions on the same account by different clients (Bankomats). which become parts of the same top-level transaction. In such cases, subtransactions are necessary for synchronization.

6.6.2 A Nested Transaction Example

Having explained the outer structure of a session with Bankomat, we now look into nested transactions. We emphasize that in this example, the terms "transaction" and "subtransaction" are relative. "Transaction" does not imply top-level transaction. Rather, it refers to the transaction under consideration. In general, we use "transaction" to refer to a transaction in its own context. and "subtransaction" or parent transaction when related to other transactions.

Let us consider the *transfer* operation in figure 6.14 as an example. The first program, Basic-Transfer, transfers a certain amount from an account to another (the program fragment is included

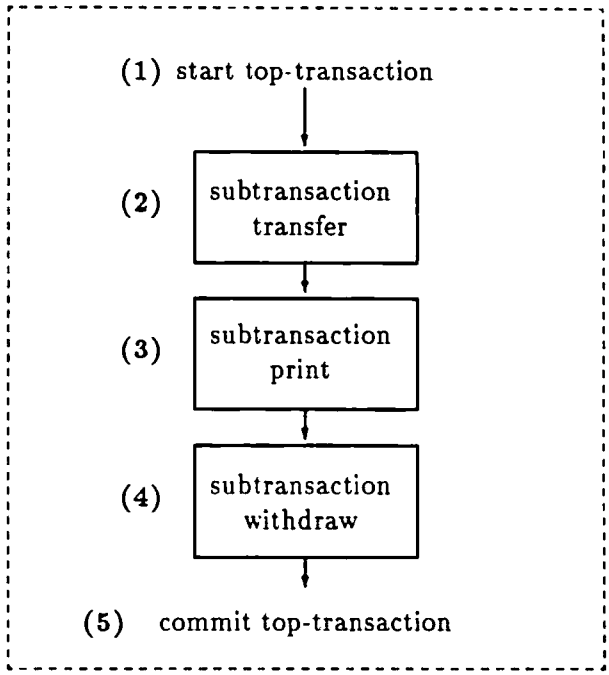

Figure 6.14: Schematic Sample Session

in appendix figure A.4). To simplify the reading, instead of the program we use a schematic illustration in this section. Figure 6.15 shows two concurrent processes, the first decrements from one account, and the second increments the other account. No subtransactions are necessary in the procedure BasicTransfer, since either both succeed and the transaction commits, or one fails and the whole transaction aborts.

Although BasicTransfer does the job, it does not offer the best possible service. For example, a customer may try to transfer $100 from checking to Visa account. If the customer does not have $100 in the checking account, the transfer fails, even though there may be thousands of dollars in the savings account. To include all three customer accounts into consideration, a new procedure –SmartTransfer– was written. Figure 6.16 illustrates SmartTransfer, which is described in detail by the program fragment in appendix figure A.6. For concreteness, instead of variables, we use the example of a transfer from checking to Visa account. Compared to BasicTransfer, the main improvement is that SmartTransfer will obtain the amount from savings, in case decrementing the checking account fails.

In the first place, there are three transactions in figure 6.16, each enclosed in a dashed box. The whole SmartTransfer is a transaction, which runs as a subtransaction of the entire session. In addition, SmartTransfer is parent transaction for the other two. We have seen BasicTransfer in figure 6.15. BasicDecrement is a subtransaction that decrements the amount from the specified account (program fragment in appendix figure A.5). Although normal implementations of a simple resource as an integer account would be atomic data types, we use a subtransaction to make a

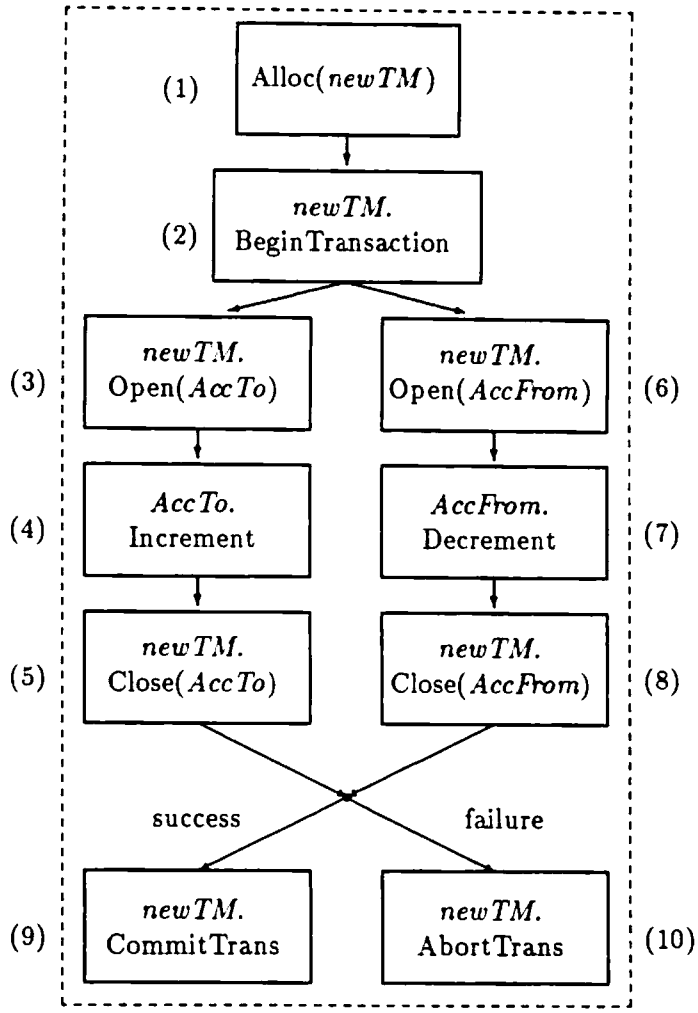

Figure 6.15: Procedure BasicTransfer

point. Namely, in terms of concurrency control and crash recovery, ERMS makes no assumptions on the resources it controls. Since we use the checking account again in the BasicTransfer subtransaction, if BasicDecrement fails we expect the checking account to be returned to its original state.

In contrast, the *Visa*.Increment operation does not have to be enclosed in a subtransaction, although it could have been, just like BasicDecrement. Since the Visa account is not invoked anywhere else in the transaction, and SmartTransfer commits if and only if both Decrement and Increment succeed, any temporary inconsistency in Visa will not affect SmartTransfer.

6.6.3 ERMS Features in SmartTransfer

The above example appears so simple that some useful ERMS features might have been overlooked by a casual reader. First, client programs only deal with resources using string names. There is no mention of object location, or the number of copies of a resource. Although there is the overhead of opening a resource, once the resource is open the client invokes the resource object directly, using its capability. This situation is similar to systems like Unix.

Second, BasicDecrement and BasicTransfer are building blocks that can run either as top-level transactions or subtransactions without modification or recompilation. For example, the Bankomat interface has an option that bypasses the enclosing session transaction, making each subtransaction a top-level transaction. Consequently, BasicTransfer may be used at three different levels: top-level transaction, subtransaction of the session transaction (as the transfer operation in figure 6.14), or a second-level subtransaction (with SmartTransfer as the transfer operation in figure 6.14).

Third, as we have mentioned in the previous section, the recovery of account state in BasicDecrement relies explicitly on nested reliability atomicity. If BasicDecrement fails, the account is expected to contain its original value. Nested concurrency atomicity is illustrated by a scenario in figure 6.17. In this scenario, a second Bankomat runs at the same time as the first one, introducing concurrent subtransactions.

In this EdenBank application, there is a dedicated transaction manager for each account name, so both Bankomats connect to the same transaction manager when they receive the same account name. All subtransactions controlled by the same transaction manager are parts of the same parent transaction. Since the first Bankomat that started the top-level transaction will also commit the top-level transaction, there must be a way to synchronize between the different Bankomats. The synchronization is done through a dummy subtransaction, which is started at the beginning of the session and committed at the end. Since the top-level transaction manager does not finish the commit protocol until all child transactions have terminated, the dummy synchronization subtransaction assures the completion of the Bankomat 2 session.

This set-up introduces a user-interface problem. Specifically, a decision to abort by Bankomat 1 would also affect Bankomat 2. An option to allow Bankomat 2 start a new top-level transaction will solve the problem. However, as a result, less sharing will be allowed between Bankomat 1 and Bankomat 2.

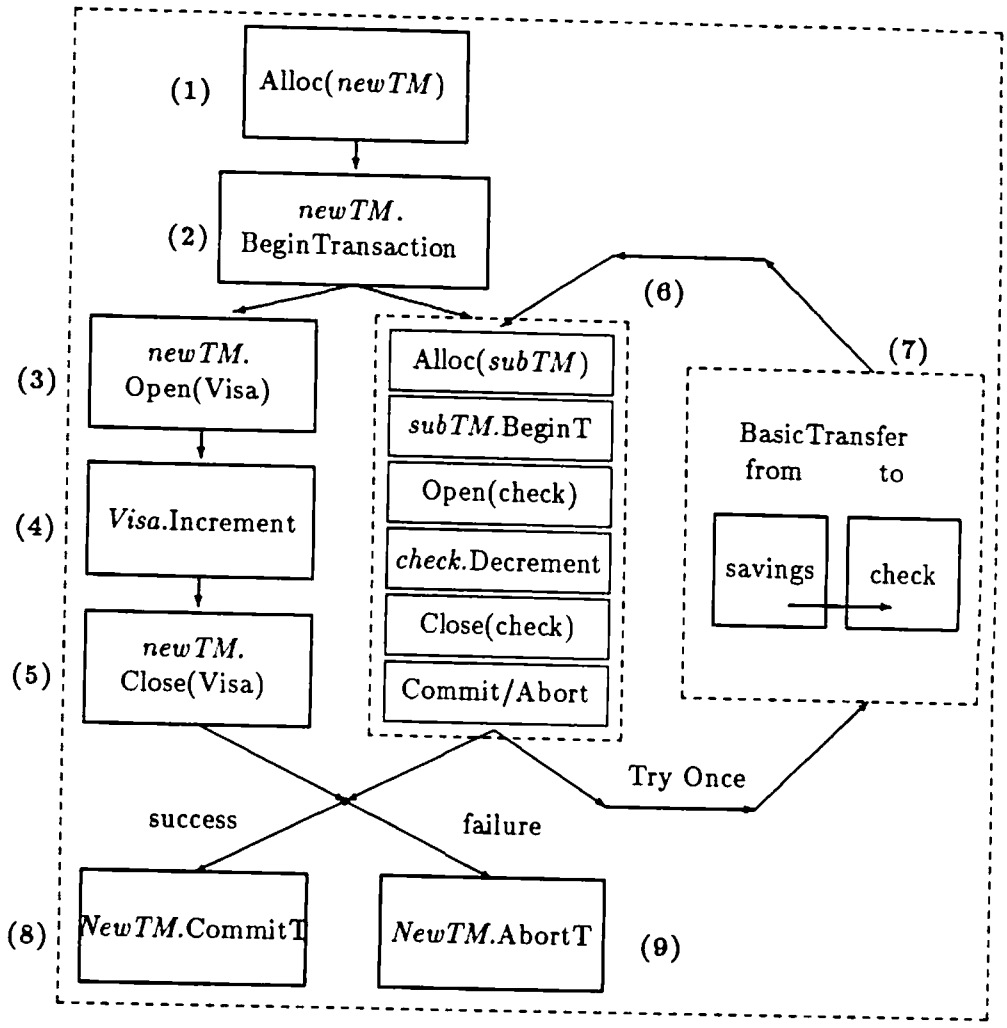

Figure 6.16: Procedure SmartTransfer, checking to Visa Example

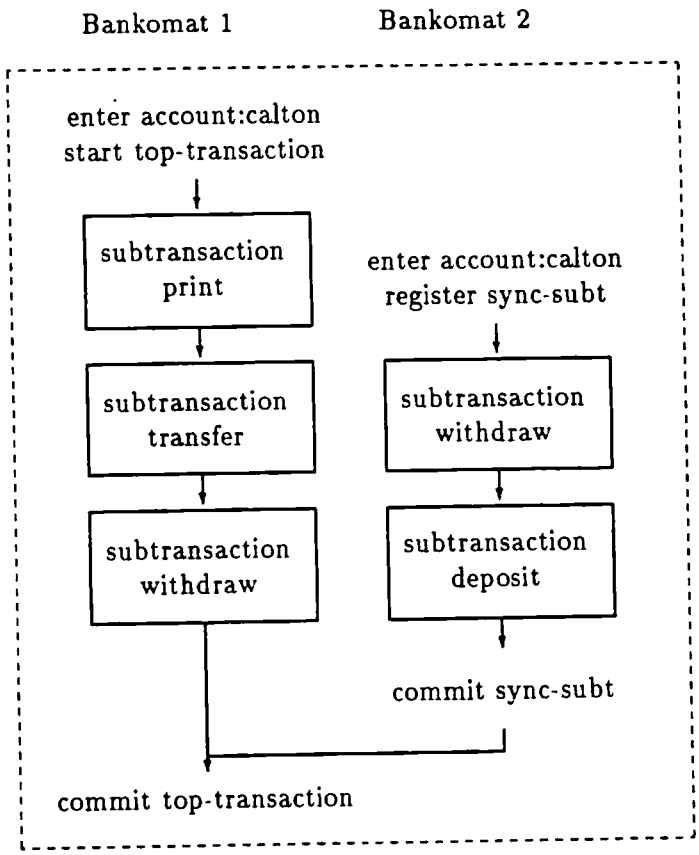

Figure 6.17: Concurrent Nested Transactions

6.6.4 ERMS Actions Behind the Scenes

In figure 6.16 we have seen an example of ERMS use. Now we describe how the ETM provides concurrency atomicity and reliability atomicity both of which are transparent to the client. The following description is based on the program in appendix figure A.6, which provides more details than figure 6.16; for instance, the parameters for invocations are described only in the program fragment.

The subtransaction to make the transfer is the procedure SmartTransfer in the Bankomat client. The first operation ((1) in figure 6.16 and program in figure A.6) in the client is to allocate the instance of ETM corresponding to the account name, and then the client sends it the BeginTransaction invocation (2). At this time, the ETM's TreeManager performs initialization according to the TimeOutPeriod and Duration parameters. First, it knows the expected time before which it should commit (TimeOutPeriod). Second, it determines whether it is a long-term transaction. Long-term transaction ETMs checkpoint their state to stable storage whenever a resource is opened or closed, so they can resume a transaction even after a crash. In this example, the ETM remains volatile. Third, the TreeManager invokes the SessionTM to tell it about the TimeOutPeriod. Finally, the TreeManager invokes the SessionTM to register as its child.

After the ETM returns from BeginTransaction, it is ready to accept requests for resource access. The Bankomat starts two parallel processes, one to increment "alice/Visa", and the other to decrement "alice/checking". We will describe the actions behind the simpler process (increment) first. ETM's ResourceManager in response to OpenResource –(3)– will first try to obtain a lock from LockTable. Since this is the first time the resource "alice/Visa" is requested, the LockTable asks the SessionTM for the lock, which in its turn, requests the lock from System Lock Manager. If there are no conflicts, the System Lock Manager gives out the lock, and SessionTM does the same. Otherwise the lock is denied and the OpenResource invocation fails. Once the lock is granted, the ResourceManager tries to Lookup the resource in RepDirTable. Since this is the first time, RepDirTable receives the capability for resource "alice/Visa" from SessionTM, passed by R2D2. This is analogous to LockTable. When the result is back, RepDirTable puts the entry in its own mapping and gives it to ResourceManager. Note that RepDirTable.Lookup and LockTable.LockName may proceed in parallel in order to achieve better performance. Finally, the ResourceManager takes the next parameter, WriteNew, which means that a new version must be created. Therefore it invokes "alice/Visa" asking it to make a copy of itself, returning the copy's capability in *AccountTo*.

The client manipulates the resources directly, so any invocations, like *Visa*.Increment –(4)–, are sent to *AccountTo*. When the client has made all the invocations, since increment normally succeeds, the resource is closed (5). The new version of "alice/Visa" substitutes the old version in DirMap, and the lock in LockMap released. Note that R2D2 continues to hold the old version of "alice/Visa", and *newTM* continues to hold its lock in SessionTM, which holds the lock in the System Lock Manager. Thus, process one ends successfully.

Process two is similar to process one, except that the Decrement operation is enclosed in a subtransaction (6). In the subtransaction BasicDecrement, described in the program fragment in appendix figure A.5, the actions are similar to process one. The resource "alice/checking" is opened by its own ETM. The locks and resource capabilities are obtained from its parent

transaction manager (in this case, *newTM*) instead of SessionTM, or System Lock Manager and R2D2. If the Decrement invocation succeeds, it is closed and the subtransaction BasicDecrement and process two terminate with success.

However, if "alice/checking" fails, EdenBank does not charge $10 for insufficient funds. Rather, Bankomat start a subtransaction (7) to attempt to move money from "alice/savings" to "alice/checking". The procedure BasicTransfer described in appendix figure A.4 simply transfers Amount from "alice/savings" to "alice/checking". If the subtransaction succeeds, we try BasicDecrement again. Since BasicTransfer is a subtransaction, its ETM will obtain its locks and resource capabilities from the parent transaction manager (*newTM*).

If everything fails, (for example, no money in "alice/checking" or "alice/savings") the client asks *newTM* to AbortTransaction (9). The TreeManager first checkpoints the abort record, making sure no other outcome is possible. Then the LockTable is asked to release the locks it is holding in the parent.

If both processes terminate successfully, the client commits the transaction by sending the CommitTransaction invocation (8) to the ETM. If the updated resources are replicated, the right number of copies must be made and distributed (maybe to specific nodes) in the network. The replicas are made and distributed in the background between the close (with the AdoptNew parameter) and commit. Although this preprocessing may waste some effort if the transaction aborts, it saves time during commit. ETM uses the Regeneration method, described in chapter 4, to update multiple copies of replicated resources.

The first step in CommitTransaction is the TreeManager checkpointing the ETM, including the commit record, the DirMap, and LockMap. Second, RepDirTable replaces previous versions in SessionTM with the newly committed versions of "alice/checking" and "alice/Visa" (and "alice/savings" if the subtransaction has been executed). Third, the LockTable releases all locks it held in the SessionTM. Finally, TreeManager again checkpoints, recording the completion of transaction commit.

6.7 Comparison with Previous Implementations

In this section, we compare ERMS with previously implemented nested transaction systems. Early designs [46,62,73] will be discussed in chapter 7 with the TM Tree framework.

A unique characteristic of ERMS is its ability to combine different concurrency control methods and crash recovery techniques. Moreover, ERMS uses unmodified, well-known techniques for single-level transaction systems as building blocks. In contrast, all other nested transaction proposals extend specific methods. For example, in their simplicity, ERMS locking rules differ from earlier work on nested transactions based on locking. Moss [61], Argus [56], and LOCUS [63] all consider the lock as some kind of token, which is held by one (sub)transaction at a time and inherited by the parent when the subtransaction terminates. Carefully extended locking rules are necessary to provide correct synchronization of all levels by one concurrency control. In comparison, ERMS uses the normal locking rules for the top-level, and the same rules would then recursively apply to the nested levels.

6.7.1 Argus

Argus is a language that supports nested transactions [56] in a manner similar to Moss's design. The computation model and client interface in Argus differs significantly from ERMS. First, Argus has integrated transactions into the language, with implicit transaction termination and resource access. On the other hand, ERMS adopts an explicit approach for both. Second, explicit invocations allow ERMS clients with parallel processes to be involved in several transactions (many to many). In comparison, in Argus an action corresponds to exactly one process.

There are some ERMS features not found in Argus. First, we have a uniform syntax (and semantics) for a top-level transaction and a subtransaction, so the same program can perform as a top-level transaction for one invocation, and a subtransaction for the next. In comparison, Argus requires the programmer to distinguish top-level transactions from subtransactions by using different keywords (*topaction* and *action*). Second, long-term transactions are not part of Argus. Third, Argus does not support transparent resource replication.

In compensation, Argus has some language features not supported by ERMS. Argus synchronizes object access for both internal variables and external persistent objects. ERMS relies on EPL and Concurrent Euclid to maintain program variable consistency.

6.7.2 LOCUS and Genesis

There are some important differences between the two LOCUS implementations of nested transactions [63,67,84] and ERMS. First, LOCUS nested transactions can handle only Unix (LOCUS) files; operations on directories, for example, cannot be rolled back [79]. In contrast, ERMS controls resources of any type. Second, LOCUS does not support long-term transactions. Finally, LOCUS nested transactions support is implemented as part of its kernel, while ERMS is built entirely on objects.

Some other features are comparable. Like LOCUS, ERMS relies on the kernel for object location. Unlike LOCUS, ERMS supports replication at the object level. The first implementation of LOCUS nested transactions [63] supported independent subtransaction abort, but not a uniform syntax to start and terminate top-level and nested transactions. The second implementation of LOCUS nested transactions [84], (sometimes called Genesis [67]) now offers a uniform syntax but not subtransaction failure isolation. ERMS provides both.

6.7.3 Distributed Transactions

Argus and LOCUS have been designed to support nested transactions. Another important area of research related to our work is transaction support in distributed database systems. (Readers interested in the development of distributed transactions are referred to a recent survey by Mohan [60].) In this section, we describe the nesting features of System R* [55] and TABS [77]. We have chosen R* and TABS because both have implemented some form of nested computations, although neither supports the standard model of nested transactions defined by Moss [62].

R* supports remote computations by organizing them into a tree. A process making remote requests is the parent of remote processes servicing the requests. At commit time, all processes in the tree participate and the transaction commits only if every participant agrees to commit.

Consequently, although R* computations are organized into a tree, R* does not provide failure isolation of sub-computations other than savepoints, which allow transaction restarts from selected places. Another restriction in R* is the lack of concurrency control between sub-computations of the same transaction. Should a transaction have two sub-computations running on the same node, resources are protected in a manner similar to monitors.

TABS provides more explicit nested transaction support. For example, a subtransaction is allowed to abort independently of its parent. In addition, subtransactions obtain their own locks, making simultaneous threads of control possible. However, resource sharing within a transaction is difficult, because subtransactions do not release locks before their parent commits. This restriction also limits the ways subtransactions can be used in the composition of new applications. In summary, a TABS transaction can be divided into separate concurrency units (subtransactions), but these units cannot be joined again. They terminate only at the top-level transaction termination.

6.7.4 Comparison Table

Table 6.1 compares the features of implemented nested transactions discussed in this section. In the LOCUS column, there are some rows with two remarks. For example, for failure isolation, LOCUS has "Yes/No". The first refers to the ability to allow subtransactions to abort independently of the parent in their first implementation [63]. The second refers to their second implementation [84], in which a subtransaction abort implies the failure of the enclosing top-level transaction.

Feature	LOCUS	Argus	ERMS	R*	TABS
Resource of any type	Unix file only	Yes	Yes	relations	Yes
location transparency	Yes	Yes	Yes	No	Yes
replica transparency	Yes	No	Yes	No	No
top/sub syntax transparency	No/Yes	No	Yes	N/A	Yes
failure isolation	Yes/No	Yes	Yes	savepoints	No
intra-transaction concurrency	Yes	Yes	Yes	one process per node	Yes
long-term transactions	No	No	Yes	No	No
combination of differ. CC and recovery	No	No	being implem.	No	No

Table 6.1: Comparing Implemented Systems

Chapter 7

Nesting by Composition

7.1 TM Tree

At the heart of ERMS is the organization of concurrency control and crash recovery in a tree, which is isomorphic to the tree-structure of nested transactions. We call this idea *TM Tree*, since both concurrency control and crash recovery data structures are encapsulated in the transaction manager.

In chapter 6, we have used TM Tree with specific techniques, namely two-phase locking and version-based recovery. In this section we generalize TM Tree into a design framework to include other concurrency control methods and crash recovery techniques. Of particular interest is the possibility of mixing different methods and techniques in one system. In section 7.1.1, we describe nested concurrency control and crash recovery in the TM Tree framework. In section 7.1.2, we discuss combination of different concurrency control and crash recovery methods. In section 7.1.3, we suggest some potential applications for the mixing of implementation techniques.

For simplicity of presentation, we continue to use a transaction manager for each transaction. In section 7.3, we shall describe refinements that may eliminate overhead introduced with such a simplistic design.

7.1.1 Nested Atomicity

In section 6.4.2, we have described the way concurrency control is nested in ERMS. During an OpenResource, lock requests follow a chain up the tree of transaction managers to obtain nested serialization of resource access (figure 6.6). Similarly, in section 6.4.3, we explained the way crash recovery is nested in ERMS. During an OpenResource, lookup requests are sent to parents to find the appropriate version for that OpenResource (figure 6.7).

Now we combine figures 6.6 and 6.7 into figure 7.1, which illustrates the more general case. In figure 7.1, we draw a generic concurrency control and a generic crash recovery at the top level. For the moment, we take the ERMS example, adopting two-phase locking for concurrency control and versions for crash recovery. As we have seen in section 6.4, the generic OpenCR operation is LookupSet, and the generic OpenCC is LockName. However, communications between transaction managers have been simplified. The subtransaction manager simply requests that the parent transaction manager open the resource. In ERMS, the open request is equivalent

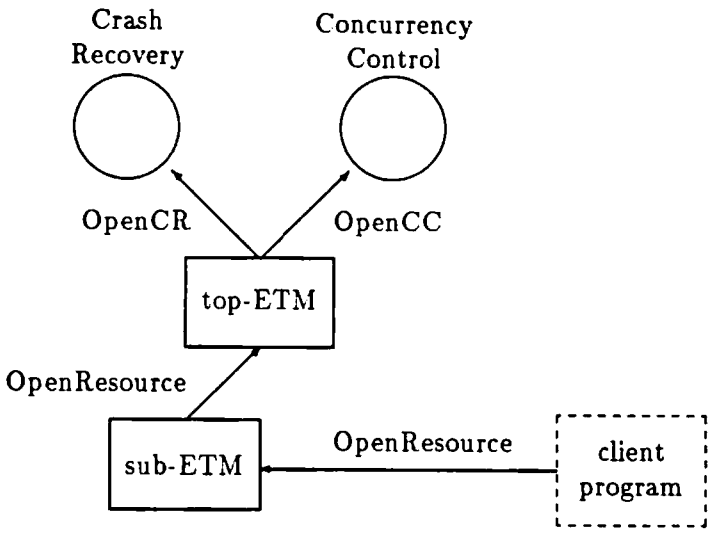

Figure 7.1: Nested OpenResource

to a LockName and Lookup. More generally, subtransaction managers can communicate with their parent in a way that is independent of particular concurrency control and crash recovery techniques.

A natural question is: "With this technique-independent interface, are we able to use other techniques in our implementation?" The answer is yes.

Let us start by adopting timestamps for concurrency control instead of locking. We keep the version-based recovery since this is the natural combination. At BeginTransaction, the transaction manager receives the transaction's timestamp (TID). Each resource has several versions (V_i); each version has its last-read and last-write timestamps, denoted by $TS_r(V_i)$ and $TS_w(V_i)$. At OpenResource, the transaction manager selects the latest version such that $TS_w(V_i)$ is less than TID. Intuitively, that version is the most recent version for the transaction's time.

If the resource has been opened for updates, a new version is created bearing the TID as its last-write timestamp. At CloseResource, this new version is stored in the transaction manager's mapping for crash recovery. During the transaction, the same resource may be opened again, creating another version. The most recent version is made public only at CommitTransaction. After the transaction manager has checked that no new versions were inserted between $TS_w(V_i)$ and TID, the new version is inserted into the resource's list of versions and made visible to other transactions. In case of AbortTransaction, the new version is simply deleted and forgotten. For illustration, actions taken by different concurrency control methods at the operations described above are summarized informally in table 7.1.

Several concurrency control methods based on dynamic timestamp intervals have been implemented for an experimental study comparing their performance, which will be reported elsewhere [66]. Unfortunately, this particular implementation did not separate clients from the transaction managers, making the adaptation to ERMS more difficult. At the time of this writing, concurrency control methods other than locking have not been incorporated into ERMS.

	Strict Two-Phase Lock	Timestamps	Optimistic Conc.Contr.
BeginTrans.	—	get TID	—
OpenCC	Lock	read V_i with max $\{i \mid TS(V_i) < TID\}$	—
CloseCC	—	—	—
CommitTrans.	Unlock	write V_{TID} if $\nexists V_k$ s.t. $V_i < V_k < V_{TID}$	check serializability
AbortTrans.	Unlock	—	—

Table 7.1: Illustration: Nested Concurrency Control Actions

From recovery techniques known in the literature [11], so far we have only discussed version-based systems. Now we turn to recovery methods that use nested log records. As explained in section 2.3, logs contain the same amount of information as versions, but recovery using logs requires more work. The recovery mechanism must scan the log, look for data relevant to the interrupted transactions, and take appropriate corrective actions on the database to restore its consistency. For example, operations from aborted transactions must be undone, back to the state at the beginning of those transactions.

Usually, a log record contains two parts. The first part identifies the transaction writing the record, and the second part contains recovery data. To recover nested transactions properly, the log must include enough information to allow the reconstruction of transaction hierarchy. We shall argue informally that the additional information on nesting hierarchy is necessary and sufficient for correct recovery of nested transactions. Without the hierarchy information, the recovery mechanism would make permanent the results from every committed subtransaction, even those from a child of an aborted transaction, which must also be aborted. With the hierarchy, the outcome of each transaction can be resolved correctly. Since the results of committed subtransactions remain conditional on the outcome of parent transactions, the recovery mechanism can restore the database to a consistent state.

The actions taken by different crash recovery techniques at the TransactionBracket and ResourceManagement operations are summarized in table 7.2. However, the table contains enough information for illustration purposes only, and the details of a recovery algorithm for log-based nested transactions remain a research problem.

7.1.2 Combining Different Techniques

All known proposals for nested transaction systems use one concurrency control method and one crash recovery technique for transactions at all levels. One unexpected result from the TM Tree

	Versions		Logging		
	Read	**Write**	**redo only**	**undo only**	**redo undo**
BeginTrans.	—	—	—	—	—
OpenCR	lookup	lookup & new version	lookup		
CloseCR	—	store new version	—	—	—
CommitTrans.	—	install new version	write everything	—	write
AbortTrans.	—	drop new version	—	undo from log	undo

Table 7.2: Summary: Nested Crash Recovery Actions

framework is that the uniformity is not necessary. Since each subtransaction manager communicates with the parent through an interface that hides the implementation, the concurrency control method and crash recovery technique used by the parent transaction manager do not have to be the same as subtransaction managers.

To illustrate this point, let us consider the example of mixed concurrency control in figure 7.2. At the system level, we adopt optimistic concurrency control, and the top-level transaction manager uses locking. When a subtransaction manager attempts to open a resource, the top-level transaction manager grants locks according to a lock compatibility table (e.g. table 2.1). Since the system level concurrency control is optimistic, the top-level transaction manager's OpenCC request translates into a null operation, not shown in the figure. At the time the top-level transaction manager attempts to commit, all resource accesses within the transaction will be checked by the system level concurrency control for conflicts.

The concurrency control mechanisms used by subtransaction managers have been omitted in the figure, and they could be any one of the many known techniques for single-level transaction serialization. Moreover, each one can adopt its own concurrency control method, independently of the other. For instance, sub-TM_1 could use locking, and sub-TM_2 timestamps.

Analogous to the variety of concurrency control methods, different crash recovery techniques may be employed. Although all proposed nested transaction mechanisms use versions to recover from crashes, as we have indicated in section 7.1.1, logging can be used, provided the nesting hierarchy information is included in the log.

There is one minor complication in mixing version-based recovery with log-based recovery. For concreteness, let us consider a simplified example of a nested transaction system running on

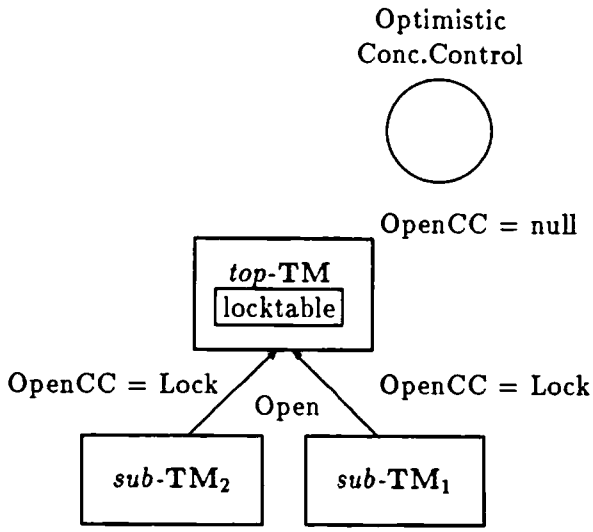

Figure 7.2: Example of Mixed Concurrency Control

a centralized database with one log device. Although each transaction may choose a different recovery technique, the recovery mechanism must have the complete nesting hierarchy information. In other words, regardless of their crash recovery method, if there are transactions relying on log-based recovery, the final results of each transaction, must be available to the log-based recovery mechanism. In particular, transactions that use versions for crash recovery may write no recovery records to the log, but they must write their commit record to the log.

7.1.3 Applications of Mixed Techniques

The ability to mix concurrency control and crash recovery techniques introduces interesting possibilities for better performance. We delineate two examples here to illustrate the point.

First, mixed concurrency control methods may allow higher effective concurrency[1] than one single method. Let us return to the example in figure 7.1. The system level concurrency control is optimistic, while the top-level transaction manager uses locking. If the transaction selects a few records from a large database, the optimistic concurrency control makes sense. However, if the two subtransactions work intensively on those few records, the probability of conflicts between the two subtransactions is high. Consequently, optimistic concurrency control in the top-level transaction manager would be counter-productive, since the subtransactions tend to conflict and abort. The use of locking at the top-level transaction manager will increase effective concurrency by avoiding aborts at that level.

Second, mixed crash recovery techniques may reduce the recovery overhead. Version-based techniques are considered "pessimistic", since they use resources to create versions during the normal processing, and require little work during recovery. In contrast, logging is "optimistic",

[1]Informally, effective concurrency may be defined as the average number of concurrent transactions making progress towards successful commit. Transactions that abort due to conflicts, for instance, do not count towards effective concurrency.

in that it reduces the normal overhead to a minimum, but need to do more work at recovery. Consequently, transactions systems with high volume always choose logging.

Now, consider a long, top-level transaction which contains a large number of fast, short subtransactions. Logging is the right choice for the short subtransactions, but may be inappropriate for the long transaction, because a large amount of log data must be processed for its recovery. So the mixing of recovery methods introduces new ways to avoid this trade-off between long recovery time and low normal overhead.

In summary, different concurrency control methods and crash recovery techniques have their own ranges of applicability and boundaries of optimal performance. The complexity of application systems and access patterns will increase as the size of databases and systems increases. Combination of different techniques within the system might improve system performance through the selection of the most adequate technique at each level.

7.2 Superdatabases

Besides the combination of different techniques in the same database system as described in section 7.1.2, an interesting alternative is to combine entire databases.

7.2.1 Distributed Databases by Composition

Past research on distributed databases [60] has produced many distributed algorithms for concurrency control and crash recovery. Despite a few exceptions like R* (from System R) and Distributed INGRESS, composition of centralized databases to form a distributed database seems to be a less explored approach.

TM Tree framework presents a systematic way to build superdatabases from element databases. The elements and the superdatabase can be either centralized or distributed. A natural combination would be centralized elements and a distributed superdatabase. In the simplest case, the element databases run the same software. As a basis for discussion, let us consider each transaction manager as a mini-database, since each includes concurrency control and crash recovery. If a requested resource is in the mini-database, client access is completely enclosed by the mini-database. Otherwise, the transaction manager obtains the resource from its parent, and "moves" the resource into the mini-database for access.

Let us take ERMS as a concrete example. If a transaction aborts, its mini-database is simply ignored. If a transaction commits, however, its mini-database is merged into the parent's mini-database. This scenario is suggested by version-based recovery, and log-based recovery achieves the same effect. The situation repeats at each level of nesting until the top level, when the system concurrency control and crash recovery take over.

To compose superdatabases out of element databases, each element must be able to imitate the mini-databases in the TM Tree. In other words, concurrency control and crash recovery in each element must be conditioned to the parent superdatabase. More concretely, an element database must support a transaction that has "committed", but which could be rolled back later because of higher level abort. This situation is not new. In section 2.2, we have described the two-phase commit protocol for distributed transactions. If the element databases support a "prepared" state

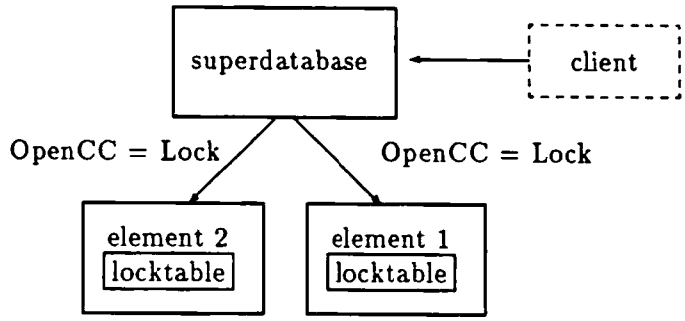

Figure 7.3: Simple Superdatabase

between the transaction commit and the final decision from the superdatabase, it seems that we can compose them.

A simple example superdatabase is shown in figure 7.3. For concreteness, we show both elements using two-phase locking, with a client accessing the superdatabase directly. To further simplify the example, we assume the two element databases –element 1 and element 2– to contain disjoint sets of resources and so they are able to keep their own lock tables. Figure 7.3 shows the moment a resource is being opened, and the superdatabase obtaining locks from both elements. At the time the client decides to commit, the superdatabase performs two-phase commit including both elements.

There are some trade-offs even in this simple example. Splitting lock tables may gain locality of access, compared to a centralized lock table with the superdatabase. However, deadlock detection becomes more complex. For example, transaction T_1 is local to element 1, and transaction T_2 is local to element 2. A deadlock may involve two supertransactions T_3 and T_4 as shown in figure 7.4. Isolatedly, each element is unable to detect this kind of deadlock. Worse yet, the deadlock cannot be detected by a simple global algorithm which looks only at inter-element dependencies; it requires the analysis of the wait-for graphs in both element databases.

7.2.2 Heterogeneous Databases by Composition

Since TM Tree framework can combine different techniques, the next natural step in the composition of databases is to compose heterogeneous databases. Past research on heterogeneous databases has focused on queries. Some examples are: MULTIBASE at the Computer Corporation of America [29,52], MERMAID at the System Development Corporation [20], and JDDBS at the Japan Information Processing Development Center [78]. They have solved the problem of translating a uniform query language into other "native" query languages. Despite the progress made in query processing, consistent update of heterogeneous databases remains a challenge [37,60].

Of the superdatabases introduced in section 7.2.1, we have assumed that the element databases are of the same kind. However, using the TM Tree framework, we can combine different implementation techniques. A simple example is shown in figure 7.5; the element databases use locking

	T_1	T_2	T_3	T_4
locked	1	2	3	4
request	3	4	2	1

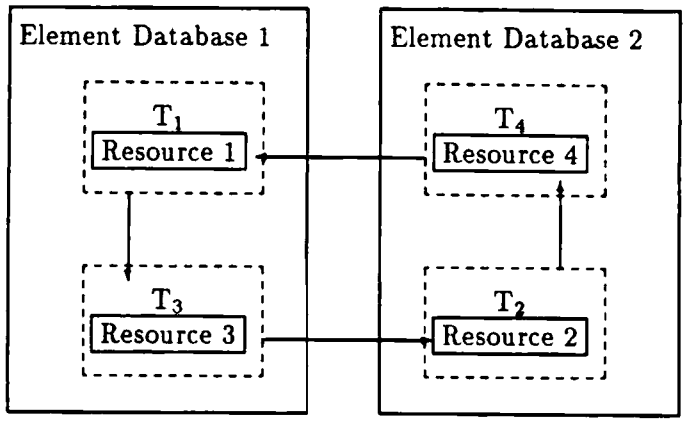

Figure 7.4: A Not-So-Simple Deadlock

and timestamps, while the superdatabase adopts optimistic concurrency control. In the example, the superdatabase has the record of all resources accesses by all transactions. Consequently, it can check for serializability of all transactions on the superdatabase.

7.3 Performance Issues

Designs derived from the TM Tree framework apparently require significant communication overhead between the TMs in the hierarchy; such is the case of ERMS. However, the performance penalty is in the implementation, rather than abstract design.

7.3.1 Reducing Communication Costs

There are two ways to look at a low-cost implementation of TM Tree designs. First, the non-object-oriented (traditional) approach would eliminate encapsulation and collapse the TM Tree into a single transaction manager per node. Consider ERMS, for example. Instead of one Eden object per ETM, we could build a Node Transaction Manager containing a tree of LockMap and DirMap. The tree of data structures substitutes the tree of Eden objects. All clients of that node send their requests to the Node Transaction Manager, accompanied of their TID. With the system-unique TID, the Node Transaction Manager finds the transaction's LockMap and DirMap, on which the operation is then performed. Invocations between subtransaction managers and parent transaction managers now translates into simple tree traversal, which is much cheaper than invocations.

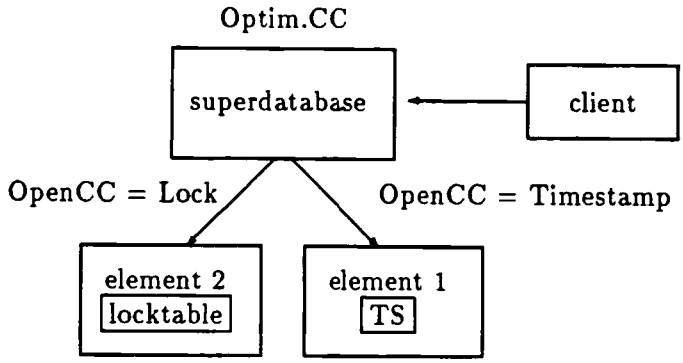

Figure 7.5: Simple Example of Heterogeneous Database

There are advantages and disadvantages in the traditional approach. For simple designs with uniform concurrency control and crash recovery, it is clear and practical. However, more general TM Tree designs with mixed concurrency control and crash recovery are much harder to put together in a single program.

The second, object-oriented approach may become a good alternative. Some new object-oriented languages and systems such as Emerald [14,15] provide efficient communications between objects, if objects are "sufficiently close". With a language like Emerald, TM Tree designs can be implemented in full generality, without heavy performance penalty.

7.3.2 Inherent Cost

In section 7.3.1, we have argued that the communication cost between transaction managers is an artifact of implementation. However, the tree structure is a fundamental part of any TM Tree design and cannot be eliminated. Compared to single-level transaction systems, the tree in the TM Tree designs requires a traversal each time concurrency control and crash recovery routines are called. Consequently, the tree traversal seems to be an inherent additional cost in nested transaction systems based on TM Tree designs.

Most tree traversal algorithms have a cost dependent on the depth of the tree. Since the TM Tree is isomorphic to the structure of nested transactions, the additional cost is proportional to the complexity of application. Although we do not have extensive experience in the use of nested transactions, current applications seem to be naturally structured into a few levels.

The first interesting problem is whether a nested transaction mechanism can be built without organizing its data structures into a tree. All existing proposals put their crash recovery information into trees. Moss has proposed a restriction (see sections 6.7.1 and 7.4.1) with which a flat lock table suffices.

Once the tree traversal overhead is accepted, the second problem is how much additional cost we must pay for the tree data structure. The analysis is not straightforward since the additional cost is not constant in each request. For example, the first time a resource is opened in a transaction, all ancestor transaction managers must obtain the appropriate lock. However,

once the resource has been opened in the transaction, the next request from any descendent subtransaction can be handled by the transaction manager, since an entry exists in the transaction manager's lock table. Consequently, the more frequently used a resource is, the less additional overhead is paid for nested atomicity.

In summary, we are optimistic about the performance of nested transaction mechanisms derived from the TM Tree framework. There are several reasons for this optimism. First, the obvious communications problem has an easy answer, with more general solutions under way. Second, the inherent problem of tree traversal is lessened by the usually shallow depth of the tree. Third, reasonably mild restrictions seem to permit the bypass of even the tree traversal. Finally, the flexibility of concurrency control and crash recovery combinations may allow a better combination than past designs.

7.4 Analyzing Earlier Designs

The most important difference between TM Tree framework and earlier designs is in generality. Many concrete designs of nested transaction mechanisms may be derived from the TM Tree framework through the selection of specific concurrency control and crash recovery techniques. In this section, we analyze earlier proposals of nested transactions using the TM Tree framework.

7.4.1 Moss

Moss [61,62] defined the standard model of nested transactions we now use. His design uses an extension of two-phase locking for concurrency control, and only a specification of what the nested crash recovery should do. His design was subsequently refined and implemented in the Argus language [56].

His extension of two-phase locking includes the notion of inherited locks, which are different from acquired locks. At commit time, a lock held by a leaf transaction is "inherited" by its parent. The parent may let other subtransactions acquire an inherited lock. However, only subtransactions holding exclusive locks can write. A parent that inherited a lock cannot access the resource.

From TM Tree point of view, a lock is a token which migrates from transaction to transaction up and down the nested transaction tree. Leaf transactions in the tree may acquire locks and access the resource, while internal transactions can only inherit locks and pass them along. Although his proposal does not organize lock tables into trees, the nesting hierarchy of transactions must be maintained somehow, so a lock being released by a subtransaction can be correctly inherited by its parent.

An interesting point is the minor restriction on the actions of the parent. To enforce the inaction of parent while in possession of inherited locks, Moss allows resource access only from the leaf transactions. The TM Tree explanation for the restriction is exactly the lack of a tree of lock tables. Since he has only one lock table, at least an additional bit per level is necessary to distinguish an inherited but unused lock from an actually "busy" lock. The former can be granted to another subtransaction, while the latter cannot. To simplify data structures, Moss disallowed direct parent access to resources, so locks are acquired and resources accessed only at the leaf

level.

For transaction roll back, Moss has specified what the nested transactions should do. Whenever an object is being written, its old content must be saved in an "associated state" with the transaction. If the transaction aborts, its associated states are used to roll back. When a subtransaction commits, its associated states are inherited by its parent to allow roll back if the parent aborts. TM Tree implements this specification by including the associated state in the transaction manager tree.

7.4.2 Reed

Reed was probably the first author to describe a detailed design to implement nested transactions [73]. In his design, he used archived, write-once versions for crash recovery. Each object is composed by a sequence of versions, each called a *possibility* when created. At transaction commit time, the possibility becomes a version and part of the object.

Reed used timestamps for concurrency control. Timestamps of transactions in the nesting hierarchy are concatenated into Pseudo Temporal Environments (PTEs), which also doubled as version names. All operations within a subtransaction are invisible from outside, since its siblings and parent cannot access its possibilities, named by its unique PTE.

The PTEs form a tree, isomorphic to the tree structure of the nested transaction hierarchy. Clearly, a TM Tree design with version-based concurrency control and crash recovery would have the transaction managers in a one-to-one correspondence with the PTEs.

Reed did not complete his implementation in the SWALLOW system.

7.4.3 Jessop

To the best of our knowledge, Jessop first suggested the use of one Transaction Manager (TM) per transaction [46]. That design, called Eden Transactional File System (EFS), is based on Bernstein and Goodman's TM/DM model [9]. In EFS, each resource has many versions under the control of its own Data Manager (DM), which synchronizes top-level access to the resource and provides crash recovery. At each level of nesting, for every resource, the transaction manager creates a copy of the data manager called workspace manager, which holds the versions created by the subtransaction. If the subtransaction commits, the workspace manager passes the committed versions back to the data manager.

Other EFS design features include resource protection based on access control lists, resource replication, a user environment, and the concept of split and join of transactions. Unlike the earlier two designs, EFS encapsulates the concurrency control in the data managers, making the transaction managers independent of any specific method. After building a simulation of EFS with immutable versions and timestamp-based concurrency control, Jessop did not implement the actual system.

TM Tree designs do not follow the TM/DM separation. Rather, the functions of the data manager are merged into the transaction manager. In Jessop's design, besides the tree of transaction managers, in a top-level transaction, the workspace managers for each resource form a tree that is a subgraph of the transaction manager tree. Conceptually coalescing these workspace

concurrency control / crash recovery	*Two-Phase Locking*	*Timestamps*	*Optimistic & others*
Version-Based	Moss [61] Argus [56] LOCUS [63,84] ERMS (chapter 6)	Reed [73] Jessop [46]	—
Logging	—	—	—

Table 7.3: Analyzing Nested Transaction Designs

managers, data managers, and transaction managers, we see the transaction manager tree emerge as the main structure in Jessop's design.

7.4.4 TM Tree Design Space

In table 6.1, we have compared the features of implemented nested transaction systems. Using the TM Tree framework, we can also compare the implementation techniques proposed by early designs. Table 7.3 shows the concurrency control mechanisms and crash recovery techniques chosen for each design, simulation, or implementation.

Several interesting observations can be derived from table 7.3. First, all current systems use versions for recovery. As we have pointed out in section 7.1.1, no detailed algorithms for log-based nested recovery is known. Second, no system uses a concurrency control method other than locking or timestamps. This is not surprising since single-level transaction systems have the same situation. Third, both TABS and R* use locking and log-based recovery, but neither provides nested reliability atomicity. Since the combination of locking and log-based recovery is considered the most efficient for single-level transaction systems, it should be a good candidate for the implementation of an efficient nested transaction mechanism.

Chapter 8

Conclusion

8.1 Summary of Contributions

This dissertation concentrates on software support to increase availability, reliability, and performance in distributed systems. From this focus, we divide our contributions into three parts: First, we have studied consistent replication to increase the availability of distributed data. Second, we have devised powerful nested transaction mechanisms to promote reliable software composition, to facilitate recovery from partial failures, and to increase concurrency at all levels of software. Third, we have built prototype systems to demonstrate the practicality of the concepts. We summarize the contributions to consistent replication in section 8.1.1, nested transactions in section 8.1.2, and systems building in section 8.1.3.

8.1.1 Replication

On data replication, we have observed the separation between hardware repair and data restoration. Concretely, we have introduced the Regeneration method, which takes advantage of this separation to update multiple copies consistently. The main idea of Regeneration is to make new copies to replace inaccessible copies. New copies maintain replicated resource consistency and reduce resource vulnerability to multiple failures. In contrast, other replication methods have limited or no ability to replace inaccessible copies.

Analytically, we have applied the k-out-of-N theory to show that Regeneration provides higher availability than other methods. To the best of our knowledge, and despite abundant literature on reliability and performance analysis, our availability analysis is the first application of the k-out-of-N theory to practical data replication methods.

However, the higher availability requires additional disk space on spare nodes to yield successful regeneration. The storage requirements are bounded by the maximum amount of inaccessible data. Updates may be allowed to complete without the additional storage, but the resource availability decreases to the same level of the Available Copies method.

Empirically, we have employed Regeneration in the implementation of R2D2, a replicated distributed directory system. R2D2 serves two purposes. First, it demonstrates the practicality of Regeneration. Second, it provides the basis for supporting replicated resources in the nested transaction mechanism we now describe.

8.1.2 Nested Transactions

We have designed and implemented a nested transaction mechanism, ERMS. Its strength is in the economy of design concepts and generality of implemented features. The key idea in ERMS design is the composition of known techniques for single-level transactions at each level to provide nested concurrency and reliability atomicity. For example, the locking rules in ERMS are the same as in many single-level transaction systems (table 2.1). In contrast, all other proposed nested transaction systems extend specific concurrency control and crash recovery techniques.

Using composition, we have introduced the TM Tree framework. Unlike previous proposals for nested transactions, TM Tree allows and suggests combinations of different concurrency control methods and crash recovery techniques. Not only is the framework a template for many nested transaction designs, but it also opens the door to more general compositions of database systems. We have just started working on a method of systematic composition of centralized databases to form distributed databases.

The ERMS implementation of nested transactions presents several important general features: Resources of any type are accessible transparently across the network. Transparent resource replication increases data availability. Subtransactions isolate partial failures to improve system reliability, while long-term transactions survive planned and unplanned shutdowns. For better performance, subtransactions execute in parallel, with consistent resource access. Finally, uniform syntax for top-level transactions and subtransactions permits easy and safe composition of super-transactions out of previously separate ones.

8.1.3 Systems

In Eden, the addition of R2D2 and ERMS addressed fault-tolerance and large scale system structuring, issues with which the Eden kernel and Eden Programming Language are less concerned. Like the Edmas mail system [1] and Eden Calendar system [42], both built on top of Eden, R2D2 and ERMS benefited from Eden's object orientation, abstract types, and location-independent objects. Unlike Edmas and Eden Calendar, R2D2 and ERMS are general-purpose tools supporting other applications. Transparent replication and nested transactions significantly augmented Eden's support for writing reliable distributed applications.

Granted, the performance of R2D2 and ERMS leaves room for improvement. That is the price we pay for building on top of Eden, an experimental prototype that is, in turn, built on top of Unix. However, it does not imply inherent inefficiency in the Regeneration algorithm or TM Tree framework, which are independent of the Eden system. For example, Regeneration reads one copy and writes all copies; for resources often read but seldom written, it should perform better than Voting.

Beyond Eden, both the Regeneration method and the TM Tree framework have wide application. The design and implementation of any replicated resource may adopt Regeneration, since most replication methods can incorporate the idea of Regeneration to improve data availability. From the TM Tree framework, many different designs and implementations of nested transactions may be derived. ERMS is a simple and general example of such a nested transaction mechanism. We have briefly analyzed the ways to improve the performance of designs derived from TM Tree framework, but only more work can confirm or contradict our optimistic outlook.

8.2 Future Work

8.2.1 Replication

Compared to hardware replication and availability, current understanding of data replication is less methodical. Although no attempt was made to survey data replication methods systematically, our analysis dividing the consistent replication problem into three parts – multiple-copy update, network partitions, and replica location – seems promising. In particular, table 3.2 captures the essential features of some important data replication proposals. A recent survey on network partitions [28] reinforces the plausibility of such classification. A taxonomy of data replication techniques will facilitate further studies that compare the range of applicability and cost of the techniques.

A more specific problem in data replication is the lack of theoretical analysis. Fortunately, there is a large body of literature on reliability theory (e.g. IEEE *Transactions on Reliability*). Nevertheless, existing theory is seldom applied to the analysis of replication algorithms. Applying reliability theory to data replication appears to be a fruitful area of research.

Another interesting phenomenon in data replication is the dichotomy between replication for availability and replication for performance (sometimes called caching). Although there are theoretical studies on the integrated analysis of system performance and availability, called *performability* [58], we have not found a replication algorithm or protocol designed for both performance and availability. Regeneration and Available Copies are both algorithms that read one copy and write "all" copies, allowing potential performance gains with exclusively local read. Research directed at replication methods for both performance and availability gains seems ready to take off.

In this dissertation, we have chosen to maintain resource consistency in the available portion of the system. In many practical systems, temporary connections between nodes form the network. In these cases, consistency can be achieved only after the elapsed time allowed eventual connection of all participant nodes. This problem increases with network size and heterogeneity. Replication techniques for eventually connected systems remain an interesting area of research.

8.2.2 Transaction Systems

Although the availability and performance potential of distributed systems have been recognized, partial degradation continues to be a serious problem. Despite productive research on fault-tolerant computing, a standard model for fault-tolerant computations remains to be found. Recovery blocks and nested transactions are possible building blocks, but considerable research separates the building blocks from a complete model.

The TM Tree framework opens a design space, allowing many different combinations of concurrency control methods and crash recovery techniques. This design space grows with the ongoing research in single-level transaction systems. A systematic exploration of the TM Tree design space in terms of applicability and performance characteristics will provide the foundation for practical implementations of nested transactions.

Built on top of Eden, ERMS is not particularly fast. Without sacrificing the generality of features, efficient implementations of nested transactions remain a challenge. In particular, nested

reliability based on logging offers the best promise, since efficient single-level transaction systems rely on logging.

A more specific problem we have left for future research is multi-level deadlock detection. Since we use known techniques in TM Tree designs, we can include a known deadlock detection algorithm with each lock table, solving the problem (except for performance). However, if the lock tables are implemented in a distributed manner, distributed deadlock detection must be employed. The interplay of mixed concurrency control methods and distribution warrants further investigation.

8.2.3 Composition

Software reuse is among the major software engineering tenets. There are many ways to reuse software, and composition is one of them. In this dissertation, we have confirmed the ease of composition offered by object-oriented systems, specifically Eden. We did not, however, remove the doubt on the performance of object-oriented systems. Efficient implementation of object-oriented systems and languages that facilitate composition is another important area of research. The Emerald project [14,15] is an example of such work.

Integrated and consistent access to a set of heterogeneous databases is useful in many ways. First, consistent access to different databases allows powerful information sharing and pooling. Second, data from an old database can be migrated gradually to new technology databases. Third, running smaller databases on specialized machines may be more cost effective than concentrating all data on one extremely fast and reliable general-purpose system.

We have described some simple cases of composition in which distributed databases and heterogeneous databases have been constructed from element databases. Our discussion has been informal and only suggests the plausibility of the approach. However, potential practical applications provide strong motivation for pursuing this line of research. Composition of element databases of the same kind is simpler and should be the first step towards the composition of heterogeneous databases.

Finally, our nested transactions have the unusual property of running the same program without recompilation at any level of nesting. Transactions with this property can be composed with other transactions at run-time, preserving the consistency of resources. We call these transactions *composable transactions*. A better understanding of composable transactions and efficient implementation will provide useful tools for the construction of reliable distributed applications.

Bibliography

[1] G.T. Almes, A.P. Black, C. Bunge, and D. Wiebe. Edmas: a locally distributed mail system. In *Proceedings of Seventh International Conference on Software Engineering*, pages 56–66, ACM/SIGSoft and IEEE Computer Society, March 1984.

[2] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–58, January 1985.

[3] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 562–570, 1976.

[4] R.E. Barlow and K.D. Heidtmann. Computing k-out-of-n system reliability. *IEEE Transactions on Reliability*, R-33(4):322–323, October 1984.

[5] D.S. Batory and W. Kim. Modeling concepts for VLSI CAD objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

[6] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic timestamp allocation for transactions in database systems. In H. J. Schneider, editor, *Distributed Data Bases*, North-Holland, 1982.

[7] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Transactions on Database Systems*, 5(2):139–156, June 1980.

[8] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[9] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[10] P.A. Bernstein and N. Goodman. The failure and recovery problem for replicated databases. In *Proceedings of Second Annual ACM Symposium on Principles of Distributed Computing*, pages 114–122, August 1983.

[11] P.A. Bernstein, N. Goodman, and V. Hadzilacos. Recovery algorithms for database systems. In *Information Processing 83*, pages 799–807, Elsevier Science Publishers B.V., 1983. Also appeared in IFIP Congress 1983 Proceedings.

[12] Ken Birman. Replication and fault-tolerance in the Isis system. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 79–86, ACM/SIGOPS, December 1985.

[13] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[14] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the First Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, September 1986.

[15] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-12(12), December 1986. To appear.

[16] A.P. Black. *The Eden Programming Language.* Technical Report 85-09-01, Department of Computer Science, University of Washington, September 1985.

[17] A.P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 181–193, ACM/SIGOPS, December 1985.

[18] B.T. Blaustein and C.W. Kaufman. Updating replicated data during communication failures. In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pages 49–58, Stockholm, August 1985.

[19] J. Bloch, D. Daniels, and A. Spector. *Weighted Voting for Directories: A Comprehensive Study.* Technical Report CMU-CS-84-114, Computer Science Department, Carnegie-Mellon University, April 1984.

[20] D. Brill, M. Templeton, and D. Yu. Distributed query processing strategies in mermaid, a frontend to data management systems. In *Proceedings of the First International Conference on Data Engineering*, 1984.

[21] S. Ceri and S. Owicki. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–129, Lawrence Berkeley Laboratory, University of California, Berkeley, February 1982.

[22] A. Chan, S. Fox, T.A. Landers, A. Nori, and D. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of SIGMOD Conference on Management of Data*, pages 184–191, June 1982.

[23] A. Chan and D. Skeen. *The Reliability Subsystem of a Distributed Database Manager.* Technical Report CCA-85-02, Computer Corporation of America, February 1986.

[24] D. Daniels and A. Spector. An algorithm for replicated directories. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 104–113, August 1983.

[25] D. Davcev and W. Burkhard. Consistency and recovery control for replicated files. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 87–96, ACM/SIGOPS, December 1985.

[26] S.B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):457–481, September 1984.

[27] S.B. Davidson. *An Optimistic Protocol for Partitioned Distributed Databases.* PhD thesis, Department of Computer Science, Princeton University, August 1982.

[28] S.B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[29] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proceedings of the Ninth International Conference on Very Large Data Bases*, October-November 1983.

[30] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of ACM*, 19(11):624–633, November 1976.

[31] S.Z. Faissol. *Operation of Distributed Database Systems Under Network Partitions.* PhD thesis, Department of Computer Science, University of California, Los Angeles, 1981.

[32] R.J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, Department of Computer Science, University of Washington, December 1985.

[33] H. Garcia-Molina, T. Allen, B. Blaustein, R.M. Chilenskas, and D.R. Ries. Data-Patch: integrating inconsistent copies of a database after a partition. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, pages 38–44, IEEE, October 1983.

[34] D.K. Gifford. *Information Storage in a Decentralized Computer System.* Technical Report CSL-81-8, Xerox PARC, March 1982. Revised version of his Ph.D. thesis.

[35] D.K. Gifford. *Violet, an Experimental Decentralized System.* Technical Report CSL-79-12, Xerox PARC, September 1979.

[36] D.K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, ACM/SIGOPS, December 1979.

[37] V. Gligor and G.L. Luckenbaugh. Interconnecting heterogeneneous database managementsystems. *Computer*, 17(1):33–43, January 1984.

[38] 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*, 13(2):223–242, June 1981.

[39] J.N. Gray. The transaction concept: virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 144–154, September 1982.

[40] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[41] M.P. Herlihy. *Replication Methods for Abstract Data Types.* PhD thesis, Massachusetts Institute of Technology, May 1984.

[42] C. Holman and G.T. Almes. *The Eden Shared Calendar System.* Technical Report 85-05-02, Department of Computer Science, University of Washington, June 1985.

[43] R.C. Holt. *Concurrent Euclid, The Unix System, and Tunis.* Addison-Wesley Publishing Company, 1983.

[44] F.S. Hsu. Reimplementing remote procedure calls. 1985. Department of Computer Science, University of Washington, M.Sc. Thesis.

[45] N. Hutchinson. *The Emerald Language and Compiler.* PhD thesis, Department of Computer Science, University of Washington, Fall 1986.

[46] W.H. Jessop, J.D. Noe, D.M. Jacobson, J-L. Baer, and C. Pu. The Eden transaction-based file system. In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, July 1982.

[47] E.B. Jul. *Distribution and Mobility in Emerald.* PhD thesis, Department of Computer Science, University of Washington, Fall 1986.

[48] W.H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys*, 13(2):149–184, June 1981.

[49] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *Transactions on Database Systems*, 6(2):213–226, June 1981.

[50] L. Lamport. Time, clocks and ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, July 1978.

[51] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. 1979. Unpublished, Xerox PARC; to appear in *CACM*.

[52] T. Landers and R.L. Rosenberg. An overview of multibase. In H.J. Schneider, editor, *Distributed Data Bases*, North Holland Publishing Company, September 1982. Proceedings of the Second International Symposium on Distributed Data Bases.

[53] E.D. Lazowska, H.M. Levy, G.T. Almes, M.J. Fischer, R.J. Fowler, and S.C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 148–159, ACM/SIGOPS, December 1981.

[54] P. Leach et al. Architecture of an integrated local area network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–857, November 1983.

[55] B. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost. Computation and communication in R*: a distributed database manager. *ACM Transactions on Computer Systems*, 2(1):24–38, February 1984.

[56] B.H. Liskov and R.W. Scheifler. Guardians and Actions: linguistic support for robust, distributed programs. In *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, pages 7–19, January 1982.

[57] Peter Ma. A look at the Eden checkpoint protocol. January 1985. Eden Internal Document.

[58] J.F. Meyer. *Unified Performance-Reliability Evaluation.* Technical Report CRL-TR-27-84, University of Michigan, Computing Research Laboratory, April 1984.

[59] T. Minoura and G. Wiederhold. Resilient Extended True-Copy Token scheme for a distributed database system. *IEEE Transactions on Software Engineering*, SE-8(3):173–189, May 1982.

[60] C. Mohan. *Tutorial: Recent Advances in Distributed Database Management.* IEEE Computer Society Press, December 1984.

[61] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Volume 1 of *Information Systems*, The MIT Press, 1985. Revised version of his Ph.D. thesis [62].

[62] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology, April 1981.

[63] E.T. Mueller, J.D. Moore, and G. Popek. A nested transaction mechanism for LOCUS. In *Proceedings of Ninth Symposium on Operating Systems Principles*, pages 71–89, ACM/SIGOPS, October 1983.

[64] S.J. Mullender and A.S. Tanenbaum. A distributed file service based on optimistic concurrency control. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 51–62, ACM/SIGOPS, December 1985.

[65] J.D. Noe, A. Proudfoot, and C. Pu. Replication in distributed systems: the Eden experience. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, ACM and IEEE/Computer Society, Dallas, November 1986.

[66] J.D. Noe and D. Wagner. Performance comparison of concurrency control methods. 1986. In preparation.

[67] T.W. Page, M.J. Weinstein, and G. Popek. Genesis: a distributed database operating system. In *Proceedings of 1985 SIGMOD International Conference on Management of Data*, pages 374–387, ACM/SIGMOD, 1985.

[68] C.H. Papadimitriou. Serializability of concurrent updates. *Journal of ACM*, 26(4):631–653, October 1979.

[69] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9:240–247, May 1983.

[70] W.H. Paxton. A client-based transaction system to maintain data integrity. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 18–23, ACM/SIGOPS, December 1979.

[71] M. Pease, R. Shostack, and L. Lamport. Reaching agreement in the presence of faults. *Journal of ACM*, 27(2):228–234, April 1980.

[72] A. Proudfoot. *Replects: data replication in the Eden System.* Technical Report 85-12-04, Department of Computer Science, University of Washington, December 1985. M.Sc. thesis.

[73] D.P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Massachusetts Institute of Technology, September 1978.

[74] D.A. Rennels. Fault-tolerant computing - concepts and examples. *IEEE Transactions on Computers*, C-33(12):1116–1129, December 1984.

[75] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[76] R.D. Schlichting and F.B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[77] A.Z. Spector, D.S. Daniels, D.J. Duchamp, J.L. Eppinger, and R. Pausch. Distributed transactions for reliable systems. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 127–146, ACM/SIGOPS, December 1985.

[78] M. Takizawa. Heterogeneneous distributed database system: jddbs. *Database Engineering Bulletin*, 6(1), March 1983.

[79] G.I. Thiel. *Partitioned Operation and Distributed Data Base Management System Catalogs.* PhD thesis, Department of Computer Science, University of California, Los Angeles, 1983.

[80] R.H. Thomas. A majority consensus approach to concurrency control. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[81] J.S.M. Verhofstadt. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):149–184, June 1978.

[82] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, pages 43–98, Princeton Unversity Press, 1956.

[83] W.E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, Massachusetts Institute of Technology, March 1984. Tech.Report MIT/LCS/TR-314.

[84] M. Weinstein, T. Page, B. Livezey, and G. Popek. Transactions and synchronization in a distributed operating systems. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 115–126, ACM/SIGOPS, December 1985.

[85] D.D. Wright. *Merging Partitioned Databases.* PhD thesis, Department of Computer Science, Cornell University, September 1983. Tech.Report No. 83-575.

[86] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessing operating system. *Communications of ACM*, 17(6):337–345, June 1974.

Appendix A

Implementation Details

A.1 Abstract Types

The concept of abstract types has not been completely formalized in EPL [16], but it has been formally incorporated into the Emerald language [15]. Informally, an abstract type is a set of invocations, supported by concrete Edentypes. Edentypes are EPL programs, and some of them are summarized in section A.2. In this section, we summarize the invocations, which define the following abstract types:

- RepDirectory – section A.1.1.
- ERMSBasic – section A.1.2.
- ERMSDebug – section A.1.3.
- TwoPhaseLock – section A.1.4.
- TransactionBracket – section A.1.5.
- ResourceManagement – section A.1.6.

A.1.1 RepDirectory

Table A.1 summarizes the invocations that define the abstract type RepDirectory. The list of main failure status codes follows:

- Some common return status codes are:
  - Success – operation succeeded.
  - IllegalString – operation refused; string name contains non-ASCII characters.
  - KernelError – operation failed; Kernel.Checkpoint failed for some reason and no change was made to the passive representation.
  - NoSuchName, NoSuchPath – operation failed; string name/path is not in R2D2.
- AddSet:

invocation	input	action	output
AddSet	newname: String capaset: Capas uid: UniqueId	insert pair into mapping	status: EdenStatus
DeleteSet	oldname: String uid: UniqueId	delete pair from mapping	status: EdenStatus
LookupSet	oldname: String uid: UniqueId	delete pair from mapping	capaset: Capas status: EdenStatus
ReplaceSet	oldname: String newset: Capas uid: UniqueId	replace capaset in mapping	status: EdenStatus
ListNames	—	list names in mapping	output: Stream status: EdenStatus

Table A.1: Abstract Type RepDirectory

  - NameAlreadyExists – insertion failed; newname is already in R2D2.

- ReplaceSet:

  - PathBlocked, NotYourLock – replacement failed; temporary lock conflict in R2D2, try again later.

- ListNames:

  - ListError – Listing failed; EdenIOStream package failed to establish listing channel.

A.1.2 ERMSBasic

Table A.2 summarizes the invocations that define the abstract type ERMSBasic. The list of main failure status codes follows:

- Some common return status codes are:

  - Success – operation succeeded.
  - KernelError – operation failed; Kernel.Checkpoint failed for some reason and no change was made to the passive representation.

- CheckpointAt

  - NoCheckSite – operation failed; specified node for checkpoint is not recognized.

invocation	input	action	output
CheckpointAt	node: Capability	checkpoint on node	status: EdenStatus
CheckpointSelf	—	checkpoint to disk	status: EdenStatus
CopySelf	—	make copy on disk	newcopy: Capability status: EdenStatus

Table A.2: Abstract Type ERMSBasic

A.1.3 ERMSDebug

invocation	input	action	output
SetCkptLimit	max: Integer	ckpt. after max updates	status: EdenStatus
SetNoiseLevel	level: Integer	write debug messages	status: EdenStatus
SetPathName	name: String copy: Integer	object name, copy #	status: EdenStatus

Table A.3: Abstract Type ERMSDebug

Table A.3 summarizes the invocations that define the abstract type ERMSDebug. The list of main failure status codes follows:

- The return status codes are:
  - Success – operation succeeded.
  - KernelError – operation failed; Kernel.Checkpoint failed for some reason and no change was made to the passive representation.

A.1.4 TwoPhaseLock

Table A.4 summarizes the invocations that define the abstract type TwoPhaseLock. The list of main failure status codes follows:

- Some common return status codes are:

invocation	input	action	output
LockName	name: String exclusive: Boolean owner: UniqueId	grant lock to owner	conflict: UniqueId status: EdenStatus
UnlockName	name: String owner: UniqueId	delete lock from table	newcopy: Capability status: EdenStatus

Table A.4: Abstract Type TwoPhaseLock

- Success – operation succeeded.

- LockName
  - NameLockedRead – exclusive lock denied; name already shared locked.
  - ExclusivelyLocked – share lock denied; name exclusively locked.
  - NameAlreadyLocked – exclusive lock denied; name already locked.

- UnlockName
  - NameNotLocked – unlock failed; name is not locked.
  - NotYourLock – unlock failed; not owner of lock.

A.1.5 TransactionBracket

invocation	input	action	output
BeginTransaction	parent: Capability longterm: Boolean expire: Timestamp	register, initialize, return tid	tid: Capability status: EdenStatus
AbortTransaction	—	release locks, clean up	status: EdenStatus
CommitTransaction	—	update parent mapping, release locks	status: EdenStatus

Table A.5: Abstract Type TransactionBracket

Table A.5 summarizes the invocations that define the abstract type TransactionBracket. The list of main failure status codes follows:

- Some common return status codes are:
  - Success – operation succeeded.
  - KernelError – operation failed; Kernel.Checkpoint failed for some reason and no change was made to the passive representation.
  - xWrongState – operation x failed; attempted operation impossible at this state; for example, BeginTransaction on a transaction already begun.
- AbortTransaction
  - LockMergeFailed – transaction aborted anyway; some locks were not released from the parent, (if top-level, then System Lock Manager).
- CommitTransaction
  - LockMergeFailed – transaction aborted; some locks were not released from the parent (if top-level, then System Lock Manager).
  - DirMergeFailed – transaction aborted; some versions were not installed in the parent (if top-level, then R2D2).
  - TimedOut – transaction aborted; time-out period expired.

A.1.6 ResourceManagement

invocation	input	action	output
OpenResource	name: String access: Pattern	section 6.3	resource: Capability status: EdenStatus
CloseResource	name: String resource: Capability do: action	save resource release locks	status: EdenStatus

Table A.6: Abstract Type ResourceManagement

Table A.6 summarizes the invocations that define the abstract type ResourceManagement. The list of main failure status codes follows:

- Some common return status codes are:
  - Success – operation succeeded.
  - IllegalString – operation refused; string name does not conform to the convention.
  - KernelError – operation failed; Kernel.Checkpoint failed for some reason and no change was made to the passive representation.

- xWrongState – operation x failed; attempted operation impossible at this state; for example, OpenResource on a transaction already aborted.

- OpenResource
  - NameLockedRead, ExclusivelyLocked, NameAlreadyLocked – open failed; unable to obtain the right lock.
  - NoSuchName, NoSuchPath – open failed; unable to find the resource name (up to R2D2).
  - AllCopiesDown – open failed; unable to find an operational copy.

- CloseResource
  - NotYourLock, NameNotLocked – close failed; resource name not open by you.
  - NoSuchName – close failed; unable to find the resource name in local DirMap.

A.2 Concrete EdenTypes

A concrete Edentype is an EPL program. Section A.2.5 contains a simple and complete example of an Edentype. When an Eden object is activated, the kernel runs the EPL program. Each Edentype has two main parts. First, invocation procedures must be written to service the invocations supported by the Edentype. Second, the rest of the program contains internal procedures and processes that execute within the program. The invocations supported by an Edentype are sometimes grouped into abstract types (section A.1). A cross reference of Edentypes with their abstract types is summarized in table A.7. The implementation of each Edentype is described in subsequent sections.

	R2D2Root	R2D2TM	ETM	RepDir
ERMSBasic	Yes	Yes	Yes	Yes
RepDirectory	Yes	Yes	Yes	Yes
TwoPhaseLock	Yes	Yes	Yes	No
TransactionBracket	No	No	Yes	No
ResourceManagement	No	No	Yes	No

Table A.7: Abstract Types Supported by Concrete Types

A.2.1 R2D2Root

- Function: R2D2 root object; receives and forwards R2D2 requests.
- Size: 1964 lines of EPL.
- Supports 20 invocations:
  - Abstract Type ERMSBasic: CheckpointAt, CheckpointSelf, CopySelf.
  - Abstract Type ERMSDebug: SetCkptLimit, SetNoiseLevel, SetPathName.
  - Abstract Type RepDirectory: AddSet, DeleteSet, ReplaceSet, LookupSet, ListNames.
  - Four invocations for upward compatibility with earlier directories: Lookup, LookupEntry, NextEntry, List.
  - Three invocation for regeneration: SwitchRequest, SwitchResult, SwitchCancel.
  - Two invocations for top-level directory management: LookupTop, UpdateTop.
- The utility modules used in R2D2Root are:
  - CheckName (section A.3.1).
  - Ckpt (section A.3.2).
  - LockMap (section A.3.5).
  - LocationMgr (section A.3.4).
  - RoottmMgr (section A.3.6).
  - SWIMap (section A.3.7).

A.2.2 R2D2TM

- Function: R2D2 transaction manager in Access Structure.
- Size: 2712 lines of EPL.
- Supports 16 invocations:
  - Abstract Type ERMSBasic: CheckpointAt, CheckpointSelf, CopySelf.
  - Abstract Type ERMSDebug: SetCkptLimit, SetNoiseLevel, SetPathName.
  - Abstract Type RepDirectory: AddSet, DeleteSet, ReplaceSet, LookupSet, ListNames.
  - Four invocations for upward compatibility with earlier directories: Lookup, LookupEntry, NextEntry, List.
  - One invocation for caching the top-level directory: UpdateTop.

- The utility modules used in R2D2TM are:
  - CheckName (section A.3.1).
  - Ckpt (section A.3.2).
  - LocationMgr (section A.3.4).

A.2.3 ETM

- Function: ERMS transaction manager.
- Size: 2574 lines of EPL.
- Supports: 27 invocations:
  - Abstract Type ERMSBasic: CheckpointAt, CheckpointSelf, CopySelf.
  - Abstract Type ERMSDebug: SetCkptLimit, SetNoiseLevel, SetPathName.
  - Abstract Type TwoPhaseLock: LockName, UnlockName.
  - Abstract Type TransactionBracket: BeginTransaction, AbortTransaction, CommitTransaction.
  - Abstract Type ResourceManagement: OpenResource, CloseResource.
  - Abstract Type RepDirectory: AddSet, DeleteSet, ReplaceSet, LookupSet, ListNames.
  - Four invocations for upward compatibility with earlier directories: Lookup, LookupEntry, NextEntry, List.
  - Five invocations for internal management: RegisterTM, ReleaseTM, QueryTM, ListLock, ListChildren.
- The utility modules used in ETM are:
  - CheckName (section A.3.1).
  - Ckpt (section A.3.2).
  - DirMap (section A.3.3).
  - LockMap (section A.3.5).
  - LocationMgr (section A.3.4).

A.2.4 RepDir

- Function: Mapping from String names into corresponding sets of capabilities; concretely, elements of R2D2 Core Structure.
- Size: 1654 lines of EPL.
- Supports 15 invocations:

- Abstract Type ERMSBasic: CheckpointAt, CheckpointSelf, CopySelf.
- Abstract Type ERMSDebug: SetCkptLimit, SetNoiseLevel, SetPathName.
- Abstract Type RepDirectory: AddSet, DeleteSet, ReplaceSet, LookupSet, ListNames.
- Four invocations for upward compatibility with earlier directories: Lookup, LookupEntry, NextEntry, List.

- The utility modules used in RepDir are:
  - CheckName (section A.3.1).
  - Ckpt (section A.3.2).
  - DirMap (section A.3.3).
  - TIDMap (section A.3.9), including tdmap (section A.3.8).

A.2.5 EdenInteger

- Function: Example Eden object implementing integer.
- Size: 64 lines of EPL.
- Supports 2 invocations:
  - One to read the integer value: Read.
  - One to alter the integer value: Add.
- No imported utility modules.

The Edentype EdenInteger does not use utility modules. Its purpose is to serve as a simple example of an Eden object. The entirety of EdenInteger EPL code is included in figure A.1. The abstraction which EdenInteger implements is that of a signed integer number. The invocations it supports are Read, which returns the current value of the integer, and Add, which adds an integer to the current value.

From the Eden object programmer point of view, one major accomplishment of EPL is language support for invocations. Eden invocations have the syntax and semantics of Remote Procedure Calls, but Eden kernel primitives are message-oriented. EPL generates a considerable amount of code to translate between two styles of programming. For brevity, we do not list the code generated, only the word count to show the proportion of generated to programmer written code. In figure A.2, which is a direct output from the Unix "wc" program, the file **EdenInt.epl** is the source program listed in figure A.1. All the other files are Concurrent Euclid modules generated by the EPL translator. Excluding coincident files, the total lines of code generated is 280, which is significant compared to the 64 lines of source code. The proportion decreases as the size of programs increases. Table A.3 shows that all Edentypes listed in this appendix have more lines of code generated by EPL than those written by the programmer.

```
VAR EdenInteger : MODULE
    EXPORTS(Add, Read)
    INCLUDE '%EjectHead'
    VAR  amount: Integer := 0

INVOCATION PROCEDURE Read(callrights: EdenRights,
    VAR intval: Integer, VAR edstat: EdenStatus) =
    IMPORTS(amount)
    BEGIN intval := amount
    END Read

INVOCATION PROCEDURE Add(callrights: EdenRights,
    addval: Integer, VAR edstat: EdenStatus) =
    IMPORTS(VAR amount)
    BEGIN amount := amount + addval
    END Add

PROCESS TakeInvocations
Imports(VAR Dispatcher, Kernel, CallInvocationProcedure)
BEGIN VAR Invoc : Kernel.InvkHandle
    VAR WriteOps : Dispatcher.OperationSet
    Dispatcher.MakeOperationSet("Read Add", WriteOps)
    LOOP Dispatcher.ReceiveOperation(Invoc, WriteOps)
        CallInvocationProcedure(Invoc) END LOOP
END TakeInvocations
END MODULE { EdenInteger }
```

Figure A.1: The EdenInteger Edentype

```
number of Lines   Words  Characters  File name

          144     348     5080        Cip.e
           15      45      467        Defs.e
          121     291     4453        Defs.ppe
            0       2      101        Interface.code
            9     113      569        Interface.deci
           15      45      467        Interface.text
           64     145     1830        EdenInt.epl
           64     170     1555        EdenInt.ppe
```

Figure A.2: Summary of EPL Generated Code

Edentype	Written	Generated	Proportion
EdenInteger	64	280	4.38
RepDir	1654	2860	1.73
R2D2Root	1964	3343	1.70
ETM	2574	3523	1.37
R2D2TM	2712	3068	1.13

Figure A.3: Comparison, Generated to Written Code (lines)

A.2.6 Edentype Bankomat

As part of the example application described in section 6.6, we include the program fragments of procedures BasicTransfer (figure A.4), BasicDecrement (figure A.5), and SmartTransfer (figure A.6).

```
        {A client procedure doing the simple-minded transfer. }
        PROCEDURE BasicTransfer(Amount, FromAccount, ToAccount,
                                VAR status) = BEGIN
            {Amount = amount to be transferred, FromAccount ⟹ ToAccount}
(1)     SysLockMgr.Allocate(newTM)
            {Allocate a new ETM from System Lock Manager -SLM- (newTM). }
(2)     newTM.BeginTransaction(ParentTM, TimeOutPeriod, ShortDuration)
            {ParentTM is parent TM's capability, null capability for top-level. }
            {TimeOutPeriod: transaction's max life span, after that TM aborts.}
            {ShortDuration tells the TM to bypass checkpoints at each state change. }
        COBEGIN     {Processes are not subtransactions: no recovery attempted. }
                    {EPL does not support the cobegin syntax, used here for clarity, }
                    {but it provides light-weight, concurrent processes. }
        BEGIN   {First parallel process: increment ToAccount. }
(3)         newTM.OpenResource(ToAccount, WriteNew, AccTo)
                    {Make a new copy of the most recent version of ToAccount; }
                    {lock the resource name ToAccount in System Lock Manager; }
                    {and return the capability of the copy in AccTo.}
(4)         AccTo.Increment(Amount, status)
                    {The client manipulates the resource directly, once it is opened.}
(5)         newTM.CloseResource(ToAccount, AdoptNew, AccTo)
                    {End of access, install the new version.}
        END     {Of first process.}
        BEGIN   {Second parallel process: decrement FromAccount.}
(6)         newTM.OpenResource(FromAccount, WriteNew, AccFrom)
                    {Again: get a new copy in AccFrom and lock the resource.}
(7)         AccFrom.Decrement(Amount, status)
(8)         newTM.CloseResource(FromAccount, AdoptNew, AccFrom)
                        {Same as above: install the new version.}
        END     {Of second process.}
        COEND
        IF both processes succeeded THEN
(9)         newTM.CommitTransaction(status)
                    {Pass the new versions to R2D2; release locks from SLM.}
        ELSE    {Both checking and savings have insufficient funds.}
(10)        newTM.AbortTransaction(status)
                    {Do not change R2D2; just release locks from SLM.}
        END IF
        END BasicTransfer
```

Figure A.4: The BasicTransfer Procedure

```
{A client procedure doing the simple-minded decrement. }
PROCEDURE BasicDecrement(Amount, FromAccount,
                         VAR status) = BEGIN
    {Amount = amount of money to be decremented from FromAccount }
SysLockMgr.Allocate(newTM)
    {Allocate a new ETM from System Lock Manager -SLM- (newTM). }
newTM.BeginTransaction(ParentTM, TimeOutPeriod, ShortDuration)
    {ParentTM is the capability of parent ETM. }
    {TimeOutPeriod: transaction's max life span, after that TM aborts. }
    {ShortDuration tells the TM to bypass checkpoints at each state change. }
    newTM.OpenResource(FromAccount, WriteNew, AccountFrom)
        {Get a new copy in AccountFrom and the resource lock.}
    AccountFrom.Decrement(Amount, status)
    newTM.CloseResource(FromAccount, AdoptNew, AccountFrom)
            {Install the new version.}
IF success THEN
    newTM.CommitTransaction(status)
        {Pass the new versions to ParentTM; release locks.}
ELSE    {Insufficient funds.}
    newTM.AbortTransaction(status)
        {Do not change ParentTM; just release locks.}
END IF
END BasicDecrement
```

Figure A.5: The BasicDecrement Procedure

```
    {A client procedure performing operations within a transaction. }
    PROCEDURE SmartTransfer(AccountName, Amount, VAR status) = BEGIN
        {Amount = amount of money to be transferred from checking to visa.      }
        {For concreteness, AccountName is shown as "alice" in the following code.  }
(1) Kernel.Create(ETM, newTM)
        {This kernel call creates an ETM object, returning its capability in newTM. }
(2) newTM.BeginTransaction(SessionTM, TimeOutPeriod, ShortDuration)
        {SessionTM is the capability of parent TM; a null capability for top-level.}
        {TimeOutPeriod is the transaction's max life span, after which TM aborts.}
        {ShortDuration tells the TM to bypass checkpoints at each state change.}
    COBEGIN     {Each process could be a subtransaction, but not in this case. }
            {EPL does not support the cobegin syntax, (used here for clarity), }
            {but it provides light-weight, concurrent processes. }
    BEGIN   {First parallel process: increment "alice/Visa". }
(3)     newTM.OpenResource("alice/Visa", WriteNew, AccountTo)
            {Make a new copy of the most recent version of "alice/Visa" in R2D2; }
            {lock the resource name "alice/Visa" in System Lock Manager; }
            {and return the capability of the copy in AccountTo.}
(4)     AccountTo.Increment(Amount, status)
            {The client manipulates the resource directly, once it is opened.}
(5)     newTM.CloseResource("alice/Visa", AdoptNew, AccountTo)
            {End of access, install the new version.}
    END     {First process.}
    BEGIN   {Second parallel process: decrement the checking.}
(6)     BasicDecrement(AccountFrom, status)
        IF status NOT = "Success"
        THEN    {"alice/checking" does not have sufficient funds.}
(7)         BasicTransfer(Amount, "alice/savings", newTM, status);
                {BasicTransfer is a nested transaction, }
                {moving Amount from "alice/savings"into "alice/checking".}
            IF status = "Success"
            THEN    BasicDecrement(AccountFrom, status) END IF
        END IF
    END     {Second process.}
    COEND
    IF both processes succeeded THEN
(8)     newTM.CommitTransaction(status)
            {Pass the new versions to R2D2; release locks in System LockMgr.}
    ELSE    {Both checking and savings have insufficient funds.}
(9)     newTM.AbortTransaction(status)
            {Do not change R2D2; just release locks from System Lock Manager.}
    END IF
    END SmartTransfer
```

Figure A.6: The SmartTransfer Procedure

A.3 Utility Modules

All the utility modules were written in Concurrent Euclid, and contain a fair amount of comments, debugging code, and blank lines. Table A.8 contains a cross reference of Edentypes with utility modules they use.

	R2D2Root	R2D2TM	ETM	RepDir
CheckName	Yes	Yes	Yes	Yes
Ckpt	Yes	Yes	Yes	Yes
DirMap	No	No	Yes	Yes
LockMap	Yes	No	Yes	No
TIDMap	No	No	No	Yes
tdmap	No	No	Yes	No

Table A.8: Concrete Edentypes and Utility Modules

A.3.1 CheckName

- Function: validity check and interpretation of string names.
- Use: check string names in RepDir and other places.
- Size: 219 lines of CE code.
- No data structures.
- 7 exported operations.
  - Three name checking operations: PathIllegal, Illegal, UIDLess.
  - Three name interpretation operations: Extractor, RLExtractor, NestingNumber.
  - One housekeeping function: UIDMakeNull.
- 1 imported module.
  - Ckpt (section A.3.2): debug and identification module.

A.3.2 Ckpt

- Function: Edentype debugging and identification module.
- Use: help debug every Edentype and utility module.
- Size: 559 lines of CE code.
- Main data structures: several "global" variables such as trace level and object name.
- 23 exported operations.
  - Ten debug operations: ErrorLevel, SetErrorLevel, CheckpointLimit, SetCheckLimit, CheckLimit, NeedCheckpoint, NeedCheck, SetNeedCheck, PathName, SetPathName.
  - Five printing operations: PrintCapa, PrintTID3, PrintXString, PrintDur, PrintEFSError.
  - Five conversion and macro operations: XStr2Char, CapasMakeNull, NullTID, MakeVM, FreeNStr, InitNStr.
  - Two housekeeping functions: PRtoAF, AFtoPR.
- No imported modules.

A.3.3 DirMap

- Function: mapping of string names into sets of capabilities.
- Use: implements the mapping in RepDir (R2D2) and ETM (ERMS).
- Size: 544 lines of CE code.
- Main data structures: a linked list, each element with a string name and a set of capabilities.
- 14 exported operations.
  - Four name manipulation operations: NameFind, NameInsert, NameDelete, NameReplace.
  - Six map query operations: MapList, MapListNames, MapNext, MapLast, MapBiggest, MapTranslate.
  - Four housekeeping functions: MapInitialize, MapPrintState, PRtoAF, AFtoPR.
- 1 imported module.
  - Ckpt (section A.3.2): debug and identification module.

A.3.4 LocationMgr

- Function: module responsible for location information, specification, and activity; concretely it provides a higher location abstraction than the kernel location-dependent primitives.
- Use: all programs concerned with location, R2D2Root, R2D2TM, ETM.
- Size: 487 lines of CE code.
- Main data structures: list of active Eden hosts and PODs.
- 12 exported operations.
  - Ten host and POD selection operations: RandomHost, RandomPOD, AcceptableHost, AcceptablePOD, PublicHost, PublicPOD, DiffHost, DiffPODSet, DiffPOD, ExtractNodes.
  - Two creation/activation operations: CreateRandom, RandomActivate.
- 1 imported module.
  - Ckpt (section A.3.2): debug and identification module.

A.3.5 LockMap

- Function: mapping of string names (resources) into locks (type of lock and lock holders).
- Use: in the implementation of the abstract type LockTable.
- Size: 1017 lines of CE code.
- Main data structures: a two-dimensional linked list. Each element of the outer list contains a TID, type of lock, and a list of lock holders.
- 14 exported operations.
  - Three name entry manipulation operations: NameFind, NameInsert, NameDelete.
  - Three lock holder manipulation operations: OwnerFind, OwnerInsert, OwnerDelete.
  - Four map query operations: MapList, MapNext, MapTranslate, OwnerTranslate.
  - Four housekeeping functions: MapInitialize, MapPrintState, PRtoAF, AFtoPR.
- 1 imported module.
  - Ckpt (section A.3.2): debug and identification module.

A.3.6 RoottmMgr

- Function: module responsible for R2D2TM creation, allocation, activation, and reuse; also remembers the top-level RepDirs.
- Use: in R2D2Root to manage the instances of R2D2TM and top-level RepDirs ('/').
- Size: 488 lines of CE code.
- Main data structures: list of available R2D2TMs, each element containing a capability, and whether it has been allocated.
- 8 exported operations.
  - Two top-level RepDir management operations: LookupTop, ReplaceTop.
  - Three R2D2TM management operations: AllocateTM, IPAllocateTM, IPDeallocateTM.
  - Three housekeeping operations: TMInitialize, TMAFtoPR, TMPRtoAF.
- 1 imported module.
  - Ckpt (section A.3.2): debug and identification module.

A.3.7 SWIMap

- Function: mapping of string names (RepDir names) into locks (type of lock and lock holders).
- Use: R2D2Root has a SWIMap to synchronize regeneration operations within it, in parallel to normal updates.
- Size: 1044 lines of CE code.
- Main data structures: identical to LockMap.
- 14 exported operations, identical to LockMap above. (Concurrent Euclid does not support "generic" modules, so if two instances of a module are needed, two must be declared.)
- 1 imported module.
  - Ckpt (section A.3.2): debug and identification module.

A.3.8 tdmap

- Function: mapping of string names (invocation UIDs) into invocation status.
- Use: borrowed from DirMap and used in TIDMap.
- Size: 536 lines of CE code.
- Main data structures: identical to DirMap.

- 7 exported operations.
  - Three name entry manipulation operations: NameFind, NameInsert, NameDelete.
  - One map query operation: MapTranslate.
  - Three housekeeping functions: MapInitialize, PRtoAF, AFtoPR.
- 1 imported module.
  - Ckpt (section A.3.2): debug and identification module.

A.3.9 TIDMap

- Function: mapping of string names (invocation UIDs) into invocation status.
- Use: provides idempotency of operations in RepDir.
- Size: 134 lines of CE code.
- Main data structures: tdmap.
- 5 exported operations.
  - Two name entry manipulation operations: NameFind, NameInsert.
  - Three housekeeping functions: MapInitialize, PRtoAF, AFtoPR.
- 2 imported modules.
  - tdmap (section A.3.8: the real mapping from name to invocation results.
  - Ckpt (section A.3.2): debug and identification module.

A.4 Measurement Samples

In this section, we include some sample measurement data we obtained from R2D2. Table A.9 contains sample data from measuring 100 LookupSet invocations as described in section 5.3.3. Table A.11 contains sample data from measuring 15 pairs of AddSet invocations, each followed by a DeleteSet invocation. The net result is the same as 30 ReplaceSet invocations. In comparison, sample data from measurements of the non-replicated directory are in tables A.10 and A.12. In these measurements, the string names used always have three levels, such as *"/users/bob/test"*.

100 LookupSet invocations			
run number	elapsed time	run number	elapsed time
1	41.44 sec	6	41.58 sec
2	41.75 sec	7	41.96 sec
3	42.03 sec	8	42.64 sec
4	41.97 sec	9	41.59 sec
5	41.54 sec	10	41.43 sec

Table A.9: Measurement Sample Data – R2D2.LookupSet

100 non-replicated Lookup invocations			
run number	elapsed time	run number	elapsed time
1	23.38 sec	6	23.08 sec
2	23.13 sec	7	23.08 sec
3	23.10 sec	8	23.11 sec
4	22.98 sec	9	23.09 sec
5	23.11 sec	10	23.10 sec

Table A.10: Measurement Sample Data – Non-Replicated Lookup

	AddSet	DeleteSet
run number	elapsed time	elapsed time
1	2.03 sec	2.22 sec
2	2.11 sec	2.10 sec
3	2.08 sec	2.32 sec
4	2.14 sec	2.30 sec
5	2.20 sec	2.13 sec
6	2.28 sec	2.24 sec
7	2.20 sec	2.13 sec
8	2.15 sec	2.20 sec
9	2.09 sec	2.18 sec
10	2.35 sec	2.04 sec
11	2.20 sec	2.32 sec
12	2.10 sec	2.18 sec
13	2.15 sec	2.17 sec
14	2.26 sec	2.28 sec
15	2.03 sec	2.18 sec

Table A.11: Measurement Sample Data – R2D2.Update

	Add	Delete
run number	elapsed time	elapsed time
1	1.36 sec	1.47 sec
2	1.43 sec	1.43 sec
3	1.40 sec	1.42 sec
4	1.32 sec	1.36 sec
5	1.40 sec	1.44 sec
6	1.38 sec	1.52 sec
7	1.41 sec	1.42 sec
8	1.35 sec	1.40 sec
9	1.44 sec	1.43 sec
10	1.41 sec	1.42 sec
11	1.41 sec	1.33 sec
12	1.38 sec	1.50 sec
13	1.34 sec	1.41 sec
14	1.38 sec	1.47 sec
15	1.40 sec	1.38 sec

Table A.12: Measurement Sample Data - Non-Replicated Update

Appendix B

System Programming with Objects

During the implementation of R2D2 and ERMS, we have gained some experience writing system software, such as nested transactions and transparent replication support. Since system software is not usually written using object-oriented systems and languages, our experience in Eden may be useful to other object programmers using experimental systems.

B.1 Object Composition

The primary benefit of programming in Eden is the ease of object composition. Composition is possible at two levels. Statically, modules can be linked together; dynamically, objects can be invoked. Since invocations have the same syntax and semantics of procedure calls, the main difference between the two forms of composition is in cost. Linking modules together produces larger objects, while invocations carry high run-time overhead.

We have mentioned some Eden invocation timing figures in table 5.1. To provide a rough estimate of Eden object size, the EdenInteger object described in appendix section A.2.5 occupies about 100K bytes of virtual memory when running. Obviously, the program written by the EPL programmer is small. The minimum size of an Eden object is occupied by parts of Eden kernel, utility routines, and stack/CE kernel.

The minimum size of Eden objects and the cost of invocations clearly reward a centralized style of programming, in which all objects are lumped into one to minimize both size and invocation overhead. Since the purpose of Eden is to experiment with distributed applications, we have had consciously to break up objects and distribute them. We believe that what we lost in performance was compensated by the clean composition leading to R2D2 and ERMS. The main tool of dynamic composition, abstract types, has become a central concept of the Emerald language [15].

B.2 Separating Mechanism From Implementation

Despite good support for composition of objects, Eden programmers have a relatively small number of building blocks to work with:

1. All resources are encapsulated in objects.
2. Objects communicate only via invocations.

3. The only way to store data on stable storage is checkpoint.

Since there is one and only one way to communicate with other objects or to write on disk, invocations and checkpoints have to be used, no matter how much they cost. Since these operations are kernel primitives, Eden object programmers cannot use a simpler operation even though the full generality of these primitives is not being used.

Unfortunately, Eden has only one general, expensive implementation for those general mechanisms. A high price is exacted even from simple uses. We maintain that does not have to be the case. From the point of view of object, an abstract type allows many concrete implementations, each tailored to a specific need. This is the position taken by the Emerald project [45,47], where specialized implementations provide efficiency to a general mechanism. Current results [14,15] seem encouraging.

B.3 Design for Testability

The major problem introduced by the encapsulation in object-oriented systems is debugging. When we are debugging an application involving several objects and the kernel, encapsulation may hide useful information. A typical problem in Eden is an invocation that does not return. All the participant objects and sometimes the kernel are suspects. To find the bug, we need some ways to see through the encapsulation.

Tracing is one of the oldest debugging methods. In strategic parts of a program, print statements allow the programmer to check important state changes. When debugging large programs, however, the amount of information printed may become overwhelming. Multi-level tracing allows selection at run-time of different levels of detail.

Normally, tracing statements are considered temporary additions to the program, taken out when the program is released. In the "normal" software development cycle, the tracing statements are associated only with the debugging phase. In an experimental environment like Eden, where the system software (including the kernel, the compiler, and library routines) undergo frequent changes, tracing information proved to be invaluable and a necessary part of many modules. During the debugging phase, traces contribute in the normal way. After a program has been debugged, its traces help to debug other parts of the system.

We will illustrate this point with an example. R2D2 has been operational since February 1985. In July 1985, some measurements were needed for a paper on Eden replication experience [65]. Occasionally, some objects being measured failed to return the invocations they were supposed to be servicing. This problem seemed to happen randomly, affecting different objects each time.

Object-level tracing from all objects involved showed that they execute normally, stopping only at an arbitrary invocation. Apparently the invoker sends the invocation and waits, while the invokee does not receive the message. The problem, therefore, seems to be with the kernel delivery of invocation messages. Kernel tracing revealed a message exchange loop involving the invoker and invokee's hosts and the invokee's POD. Additional tracing commands were put in the kernel to show that some kernel data structures have been corrupted after the invoked object has crashed and reactivated in another host. Corrupted data structures caused the loop.

In the above example, all levels of object tracing were used to determine that the objects did

not cause the problem, and all levels of kernel tracing were used, plus the additional ones, to identify it. The kernel bug was introduced in the then-latest kernel update, as part of a performance optimization effort, which eliminated some "redundant" data structures. Summarizing, in an experimental system like Eden, where higher level (object) programmers are expected to help debug lower level (kernel) software, we found it useful to maintain debugging tools at all levels. Tracing was simply the particular tool used in Eden.

B.4 Composed Messages

One fundamental difficulty in the debugging of composed objects is to find where the problem is. Information hiding, which facilitates interface, makes it difficult to locate the source of problems. In particular, a resource composed of many objects, such as R2D2, may span many nodes. To find and solve a specific problem requires specific knowledge of details of the system.

For example, if an AddSet request to R2D2 returns "KernelError", we may want to know which objects had difficulty checkpointing. Since a chain of invocations is involved in the request, the Eden standard return status does not provide the detailed information to locate the problem.

Since the problem above is aggravated by the number of objects in the system, a solution that limits the number of participant objects has limited utility. We have adopted a general mechanism in the spirit of object composition. The idea was borrowed from recent works in Artificial Intelligence on expert systems.

Some expert systems have an explanation facility, which gives the rationale for reaching certain conclusions. The explanation follows the "line of reasoning" and may have many levels. The implementation of the explanation mechanism varies from system to system, but the idea is to keep track of the places the program has visited, and why.

With each invocation in R2D2, a stack of strings is returned. Each object pushes its message to the stack, and the invoker analyzes the messages in the stack to determine the problem. Usually, the stack remains empty. If something does go wrong, the object simply pushes its identification[1] and the explanation –the event, its causes and effect– into the stack. All intermediate objects perceive the failure and push their own identification into the stack. Finally, the client pops the stack to retrace the events in search of the problem.

[1] Preferably a system-unique, human-readable identification, although in Eden, there is no system support for that.

Appendix C

Glossary

C.1 Eden Terms

- active form: Unix process executing an Edentype program, which can be invoked using its capability. Section 5.3.1 on page 41.
- CE: see Concurrent Euclid.
- checkpoint: Eden kernel primitive to write the object state to the passive representation. This operation is atomic, always preserving a consistent version of the passive representation. Section 5.3.1 on page 41.
- Concurrent Euclid: the concurrent version of Euclid, a Pascal derivative. Briefly introduced in section 5.1.2; for more details the reader must consult the CE book [43].
- Eden host: Unix process running the part of Eden kernel responsible for communications between kernels and objects. Section 5.3.1 on page 41.
- Eden objects: In Eden, all resources are encapsulated in objects written by object programmers. Section 1.3 on page 2 and Section 5.1.2 on page 35.
- Eden Programming Language: language used to write Edentype code. Section 5.1.2 on page 35.
- Eden system: An object-oriented distributed operating system. Section 1.3 on page 2, section 5.1.2 on page 35, and section 5.3.1 on page 41.
- Edentype: program written by Eden object programmer. Eden objects run the Edentype program of which they are instances. Section 5.1.2 on page 35.
- EPL: see Eden Programming Language.
- host: see Eden host.
- passive representation: permanent state of an object stored on stable storage.
- POD: Unix process running the part of Eden kernel that checkpoints an object's passive representation to disk. Section 5.3.1 on page 41.

C.2 R2D2 and ERMS Terms

- Access Structure: R2D2Root and R2D2TMs to access Core Structure. Section 5.1.4.
- Bankomat: Concrete Edentype, EdenBank application. Appendix section A.2.6.
- CheckName: Utility module, checking string name conventions. Appendix section A.3.1.
- Ckpt: Utility module. Appendix section A.3.2.
- Core Structure: tree of RepDir objects containing the actual mapping. Section 5.1.3.
- DirMap: Utility module, mapping string name into a set of capabilities. Appendix section A.3.3.
- EdenInteger: Concrete Edentype, example of a simple Edentype. Appendix section A.2.5.
- ERMSBasic: Abstract type; it defines the basic operations in ERMS. Appendix section A.1.2.
- ERMSDebug: Abstract type; it defines the debugging invocations in ERMS. Appendix section A.1.3.
- ETM: Concrete Edentype, ERMS Transaction Manager. Sections 6.3 and 6.4.
- LocationMgr: Utility module, location-related code. Appendix section A.3.4).
- LockMap: Utility module, mapping string names into lock holders. Appendix section A.3.5.
- LockTable: EPL module implementing TwoPhaseLock using LockMap. Appendix section 6.2.1.
- R2D2Root: Concrete Edentype, Root of R2D2, receiver of all R2D2 requests. Section 5.2.1.
- R2D2TM: Concrete Edentype, R2D2 transaction manager; it keeps the replica RepDir objects consistent. Section 5.2.2.
- RepDir: Concrete Edentype, R2D2 core structure objects, which contain the mapping from string names into capability sets. Section 5.2.3.
- RepDirectory: Abstract type; it defines the operations on mapping. Appendix section A.1.1
- RepDirTable: EPL module implementing RepDirectory using DirMap.
  Section 6.2.2.
- ResourceManagement: Abstract type; it defines the resource access delimiters. Appendix section A.1.6.
- ResourceManager: EPL module implementing ResourceManagement in ETM. Section 6.4.4.
- SLM: see System Lock Manager.
- System Lock Manager: An Eden object that implements a lock table. Section 6.2.1.

- TID: system-unique transaction identification.
- TransactionBracket: Abstract type; it defines the transaction delimiters. Appendix section A.1.5.
- TreeManager: EPL module implementing TransactionBracket in ETM. Section 6.4.4.
- TwoPhaseLock: Abstract type; it defines LockName and UnlockName. Appendix section A.1.4.