Recovery Management in QuickSilver

ROGER HASKIN, YONI MALACHI, WAYNE SAWDON,
AND GREGORY CHAN
IBM Almaden Research Center

This paper describes QuickSilver, developed at the IBM Almaden Research Center, which uses *atomic transactions* as a unified failure recovery mechanism for a client-server structured distributed system. Transactions allow failure atomicity for related activities at a single server or at a number of independent servers. Rather than bundling transaction management into a dedicated language or recoverable object manager, QuickSilver exposes the basic commit protocol and log recovery primitives, allowing clients and servers to tailor their recovery techniques to their specific needs. Servers can implement their own log recovery protocols rather than being required to use a system-defined protocol. These decisions allow servers to make their own choices to balance simplicity, efficiency, and recoverability.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File System Management—*distributed file systems*; *file organization*; *maintenance*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; *checkpoint/restart*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Commit protocol, distributed systems, recovery, transactions

1. INTRODUCTION

The last several years have seen the emergence of two trends in operating system design: *extensibility*, the ability to support new functions and machine configurations without changes to the kernel; and *distribution*, partitioning computation and data across multiple computers. The QuickSilver distributed system, being developed at the IBM Almaden Research Center, is an example of such an extensible, distributed system. It is structured as a lean kernel above which system services are implemented as processes (*servers*) communicating with other requesting processes (*clients*) via a message-passing *interprocess communication* (*IPC*) mechanism. QuickSilver is intended to provide a computing environment for various people and projects in our laboratory, and to serve as a vehicle for research in operating systems and distributed processing.

One price of extensibility and distribution, as implemented in QuickSilver, is a more complicated set of failure modes, and the consequent necessity of dealing with them. Most services provided by traditional operating systems (e.g., file,

Authors' address: IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099.

display) are intrinsic pieces of the kernel. Process state is maintained in kernel tables, and the kernel contains cleanup code (e.g., to close files, reclaim memory, and get rid of process images after hardware or software failures). QuickSilver, however, is structured according to the client-server model, and as in many systems of its type, system services are implemented by user-level processes that maintain a substantial amount of client process state. Examples of this state are the open files, screen windows, and address space belonging to a process. Failure resilience in such an environment requires that clients and servers be aware of problems involving each other. Examples of the way one would like the system to behave include having files closed and windows removed from the screen when a client terminates, and having clients see bad return codes (rather than hanging) when a file server crashes. This motivates a number of design goals:

(1) Properly written programs (especially servers) should be resilient to external process and machine failures, and should be able to recover all resources associated with failed entities.
(2) Server processes should contain their own recovery code. The kernel should not make any distinction between system service processes and normal application processes.
(3) To avoid the proliferation of *ad-hoc* recovery mechanisms, there should be a uniform system-wide architecture for recovery management.
(4) A client may invoke several independent servers to perform a set of logically related activities (a *unit of work*) that must execute *atomically* in the presence of failures, that is, either all the related activities should occur or none of them should. The recovery mechanism should support this.

In QuickSilver, recovery is based on the database notion of *atomic transactions*, which are made available as a system service to be used by other, higher-level servers. This allows meeting all the above design goals. Using transaction-based recovery as a single, system-wide recovery paradigm created numerous design problems because of the widely different recovery demands of the various QuickSilver services. The solutions to these problems will be discussed in detail below. However, we will first discuss the general problem of recovery management and consider some alternative approaches.

1.1 Recovery from System and Process Failures

The problems with recovery in a system structured according to the client-server model arise from the fact that servers in general maintain *state* on behalf of clients, and failure resilience requires that each be aware of problems involving the other. Examples of this state are the open files, screen windows, and address space belonging to a client process. Examples of the way one would like the system to behave include having files closed and windows removed from the screen when a client terminates, and having clients see bad return codes (rather than hanging) when a file server crashes.

Timeouts. A simple approach to recovery is for clients to set timeouts on their requests to servers. One problem with this is that it substantially complicates the logic of the client program. Another obvious problem is the difficulty of

choosing the correct timeout value: excessively long timeouts impair performance and usability, whereas short timeouts cause false error signals. Both communication delays and server response time can be unpredictable. A database request may time out because of a crash, but the database server might also be heavily loaded, or the request (e.g., a large join) might just take a long time to execute. False timeouts can cause inconsistencies where the client thinks a request has failed and the server thinks it has succeeded.

Connectionless protocols. Several systems have attempted to define away the consistency problems of timeout-based recovery by requiring servers to be connectionless, stateless, and idempotent [9, 22]. A client that sees a timeout for an uncompleted request has the option of retrying or of giving up. Servers keep no state or only "soft" state, such as buffers that are eventually retired by an LRU policy. We think the stateless model to be unworkable for several reasons. Some state, such as locks on open files or the contents of windows, is inherently "hard." Some services, such as graphics output to intersecting areas, require requests to be sequenced and not to be repeated. Furthermore, the server's semantics may require several client requests to be processed atomically. The client giving up in the middle of a sequence of related requests can cause loss of consistency.

Virtual circuits. Consistency and atomicity problems are partially solved by employing connection-oriented protocols, such as LU6.2 sessions [17]. Failures are detected by the communications system, which returns an out-of-band signal to both ends. Atomicity and consistency can be achieved within a virtual circuit via protocols built on top of it. The primary limitation of virtual circuits is that they fail independently, thus multiserver atomicity cannot be directly achieved.

Some systems use hybrid recovery techniques that fall somewhere between timeouts and virtual circuits. In the V-System [9], recovery is done by detecting process failures. The kernel completes outstanding client requests to failed servers with a bad return code. Servers periodically execute **ValidPid** calls to determine the state of processes for which they are maintaining state. If the process has failed, the state is cleaned up. V has no system-defined atomic error recovery, although an architecture for implementing this at the client level via runtime library functions has been proposed [10].

Replication. Another approach to failure resilience is through replication. Clients are presented with the view of a reliable and available underlying system. Examples of systems that use replication are Locus [29], which replicates the file system; ISIS [5] and Eden [30], which replicate storage objects; and Circus [12] and Tandem [4], which replicate processes. Replication simplifies the life of clients, and eliminates the need for them to detect and recover from server failures. However, replication is too expensive to use for some system services, and does not make sense in others (e.g., display management). Furthermore, to implement replication, servers still have to be able to detect failures and coordinate their recovery. Thus, replicated systems are usually built on top of a transaction mechanism. Given our desire to have a single recovery mechanism

institutionalized in the system, we thought transactions to be the better choice of the two.

1.2 Transactions

Previous work. There is a substantial body of literature relating to transaction-based recovery in the context of single services, such as file systems [38] and databases [15]. The applicability of the concept has been explored in the context of both local [15] and distributed [19, 32] systems. More recently, there have been several experiments with using transactions as a general recovery mechanism for an operating system. Argus [21], for example, provides language constructs for recoverable shared objects, and provides underlying system facilities for implementing these constructs. TABS [34] provides transaction management as a service running under Accent [31], and allows it to be used by *data servers* to enable them to implement recoverable objects callable by Accent messages. More recently, Camelot [36] provides a similar level of function running on Mach [3]. We will defer comparing QuickSilver to these systems until later in the paper.

Recovery demands of various servers. A painful fact is that transactions, as they are normally thought of, are a rather heavyweight mechanism. Using transactions as a single, system-wide recovery paradigm depends upon being able to accommodate simple servers in an efficient way. To get a feel for this, let us examine the characteristics of a few representative servers in QuickSilver.

The simplest class of servers are those that have *volatile* internal state, such as the window manager, virtual terminal service, and address space manager (loader). For example, the contents of windows does not survive system crashes. These servers only require a signalling mechanism to inform them of client termination and failures. Often, such servers have stringent performance demands. If telling the loader to clean up an address space is expensive, command scripts will execute slowly.

A more complex class of servers manages *replicated, volatile state.* An example is the name server that other QuickSilver servers use to register their IPC addresses. To maximize availability, this server is replicated, and updates are applied atomically to all replicas. The state of each replica is volatile (i.e., not backed up in stable storage). This is conceptually similar to Synchronous Global Memory (a. k. a. delta-common storage) [13] and *troupes* [12]. Individual replicas recover by querying the internal state of a functioning replica. The exceedingly rare catastrophic simultaneous failure of all replicas is recovered from by having servers re-register themselves. Replicated volatile state uses transaction commit to provide atomicity, yielding a useful increase in availability without the expense of replicated stable storage.

The services that require the most from the recovery manager are those that manage *recoverable state*, such as QuickSilver's transaction-based distributed file system [7]. The file system uses transactions to recover from server crashes, and to detect and recover from client crashes. Furthermore, since the file system is structured as a federation of independent servers on different nodes, the transaction mechanism provides atomicity across these servers. Finally, commit

coordination and recovery can be provided atomically between the file system and other servers (e.g., database) that might exist in the future.

A final class of users are not servers at all. Long-running application programs with large data sections (e.g., simulations), whose running time may exceed the mean time between failures of a machine, require a checkpoint facility to make their state recoverable. Just as logging can be superior to shadowing in a database system [15], incrementally logging checkpoint data may be superior to dumping the entire data section to a file. Checkpointable applications use the log directly, without using commit coordination.

1.3 A Transaction-Based Recovery Manager

The QuickSilver recovery manager is implemented as a server process, and contains three primary components:

(1) **Transaction Manager.** A component that manages commit coordination by communicating with servers at its own node and with transaction managers at other nodes.
(2) **Log Manager.** A component that serves as a common recovery log both for the Transaction Manager's commit log and server's recovery data.
(3) **Deadlock Detector.** A component that detects global deadlocks and resolves them by aborting offending transactions.

Of these three components, the Transaction Manager and Log Manager have been implemented and are in use, and will be discussed in detail. The Deadlock Detector, based on a design described by Obermarck [27], has not been implemented, but is mentioned here to show where it fits into our architecture.

The basic idea behind recovery management in QuickSilver is as follows: clients and servers interact using IPC messages. Every IPC message belongs to a uniquely identified transaction, and is tagged with its transaction ID (*Tid*). Servers tag the state they maintain on behalf of a transaction with its *Tid*. IPC keeps track of all servers receiving messages belonging to a transaction, so that the Transaction Manager (TM) can include them in the commit protocol. TM's commit protocol is driven by calls from the client and servers, and by failure notifications from the kernel. Servers use the commit protocol messages as a signalling mechanism to inform them of failures, and as a synchronization mechanism for achieving atomicity. Recoverable servers call the Log Manager (LM) to store their recovery data and to recover their state after crashes.

The recovery manager has several important properties that help it address its conflicting goals of generality and efficiency. Although the remainder of this paper describes these properties in detail, they bear mentioning now:

—The recovery manager concentrates recovery functions in one place, eliminating duplicated or ad hoc recovery code in each server.
—Recovery management primitives (commit coordination, log recovery, deadlock detection) are made available directly, and servers can use them independently according to their needs.
—The transaction manager allows servers to select among several variants of the commit protocol (one-phase, two-phase). Simple servers can use a

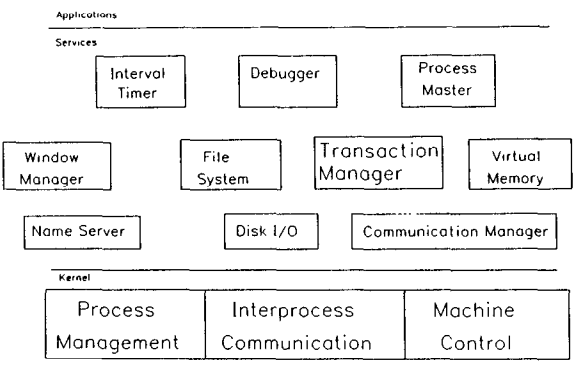

Fig. 1. QuickSilver system structure.

lightweight variant of the protocol, while recoverable servers can use full two-phase commit.

—Servers communicate with the recovery manager at their node. Recovery managers communicate among themselves over the network to perform distributed commit. This reduces the number of commit protocol network messages. Furthermore, the distributed commit protocol is optimized (e.g., when all servers at a node are one-phase or read only) to minimize log forces and network messages.

—The commit protocols support mutual dependencies among groups of servers involved in a transaction, and allows asynchronous operation of the servers.

—The log manager maintains a common log, and records are written sequentially. Synchronous log I/O is minimized, because servers can depend on TM's commit record to force their log records.

—A block-level log interface is provided for servers that generate large amounts of log traffic, minimizing the overhead of writing log records.

—Log recovery is driven by the server, not by LM. This allows servers to implement whatever recovery policy they want, and simplifies porting servers with existing log recovery techniques to the system.

2. QUICKSILVER ARCHITECTURE

A detailed discussion of the QuickSilver recovery management architecture requires some familiarity with the system architecture. Figure 1 shows the basic structure of QuickSilver. All services are implemented as processes and, with a few exceptions,[1] are loaded from the file system. Services perform both high-level functions, such as managing files and windows, and low-level device driver functions. The kernel contains three basic components: process management (creation, destruction, and dispatching), machine control (initialization, invoking interrupt handlers in user processes), and interprocess communication (IPC).

[1] The exceptions are the services used to create address spaces, processes, and to load programs, namely Process Master (loader), Virtual Memory, Disk I/O, and the File System itself.

Applications use IPC to invoke services, and services use IPC to communicate with each other.[2] Shared memory is supported among processes implementing multithreaded applications or servers, but is not used between applications and services or across any domain that might be distributed.

2.1 Interprocess Communication

QuickSilver IPC is a request-response protocol structured according to the client-server model. The basic notion is the *service*, which is a queue managed by the kernel of the node on which the service is created. Each service has a globally unique *service address* that can be used to send *requests* to the service. A service can be used for private communication between sets of processes, or can be made publicly accessible by registering it with the *name server*, which has a well-known service address. A process that wishes to handle requests to a service (a *server*), *offers* the service, establishing a binding between the service and a piece of code (the *service routine*) inside the server. When the server enters an inactive state by calling *wait*, the kernel attempts to match incoming requests to the offer, at which point the service routine will be invoked. The server can either *complete* the request (which sends the results to the client) in the service routine, or can queue the request internally and complete it later. Server processes must execute on the node at which the service was created. More than one process can offer a service, but since there is no method of directing successive requests from the same client to the same server process, the servers must cooperate to handle such requests (e.g., via a shared address space).

Client processes can issue any of three kinds of requests: *synchronous*, *asynchronous*, or *message*. Synchronous requests block the client until the server completes the request. Asynchronous requests are nonblocking and return a *request ID* that the client can use later to wait for completion. Message requests are also nonblocking, but cannot be waited on. QuickSilver IPC supports *multiple wait*: requests and/or offers can be combined into *groups*. Waiting on a group suspends the process until either a request is completed or an offer is matched to an incoming request.

QuickSilver makes several guarantees regarding the reliability of IPC: requests are not lost or duplicated, data is transferred reliably, and a particular client's requests are queued to the service in the sequence they are issued. Requests are matched to waiting offers in the order they are queued, though as mentioned they are not necessarily completed in order. If a server process terminates before completing a request, the kernel completes it with a bad return code. IPC's semantics are location-transparent in that they are guaranteed regardless of whether the request is issued to a local or remote service.

2.1.1 *Remote IPC.* When a client and server are on the same node, the kernel handles matching requests to offers, dispatching the server, and moving parameter data. When a request is made to a server on a remote node (determined by examining the service address), the kernel forwards the request to the *Communication Manager* (*CM*), a server that manages remote IPC communications (see

[2] Normally, programs issue requests by calling runtime-library stubs, which hide the details of parameter marshalling and message construction from the caller.

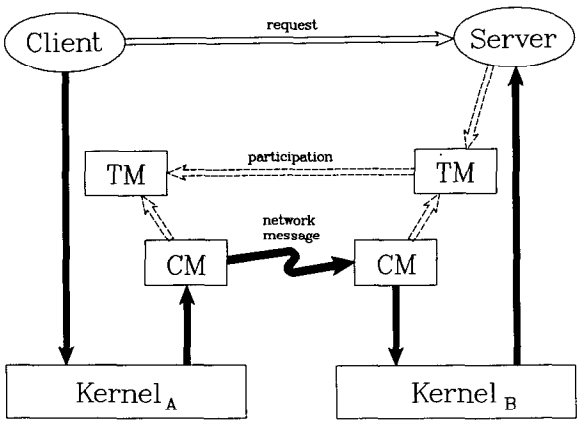

Fig. 2. QuickSilver distributed IPC.

Figure 2). CM implements the location transparent properties of IPC by managing routing, error recovery, and flow control. All IPC traffic between a pair of nodes is multiplexed over one connection maintained by the CMs on the two nodes. The CMs implement a reliable communication protocol that allows them to recover from intermittent errors (e.g., lost packets) and detect permanent ones (e.g., node or link failure), which are reported to TM. When CM detects a permanent failure of a connection to a node,[3] it causes all uncompleted requests to servers at that node to be completed with bad return codes.

3. TRANSACTION MANAGEMENT

This section describes how transactions work in QuickSilver, and the roles of clients, servers, and TMs. The commit coordination protocols are described in the next section. In QuickSilver, TM supports multisite atomicity and commit coordination. It does not manage serialization; this remains the responsibility of the servers that manage access to serializable resources. TM's knowledge of transactions is limited to those functions necessary for recovery.

Transactions are identified by a globally unique *Transaction Identifier* (*Tid*) consisting of two parts: the unique node ID of the transaction birth-site, and a sequence number that TM guarantees to be unique in each machine over time. Each IPC request in QuickSilver is made on behalf of a transaction and is tagged with its *Tid*. Run-time IPC stubs automatically tag requests they generate with a *Tid*, which defaults to one automatically created for the process when it begins, but which can be changed by the process. This allows simple clients to remain unaware of the transaction mechanism if they so choose. It is required (and enforced) that a process making a request on behalf of a transaction either be an *owner* of that transaction, or a *participant* in the transaction (both defined below). Servers tag all resources (state) they maintain on behalf of a transaction with the *Tid*. The mapping between transactions and resources enables the server to recover its state and reclaim resources when the transaction terminates.

[3] This implies either node or link failure.

Clients and servers access TM by calling run-time library stubs. These build the IPC requests to TM and return results after TM completes the request. Asynchronous variants of the stubs allow the caller to explicitly wait for the result. In the discussion below, "call" will be used to mean "send a request to."

3.1 Transaction Creation and Ownership

A process calls **Begin** to start a new transaction. TM creates the transaction, assigns it a *Tid*, and becomes the *coordinator* for the transaction. The caller becomes the transaction's *owner*. Ownership conveys the right to issue requests on behalf of the transaction and to call **Commit** or **Abort**. Clients can, if they wish, own and issue requests on behalf of any number of transactions.

The **ChangeOwner** call transfers ownership of a transaction to a different process. For example, the Process Master creates a new process, creates its default transaction, transfers ownership to the new process, and finally starts the process. Ownership spans the interval between the **Begin** or **ChangeOwner** call and the **Commit** or **Abort** call.

3.2 Participation in Transactions

When a server offers its service, it declares whether it is *stateless*, *volatile*, or *recoverable*. When a volatile or recoverable server receives a request made on behalf of a transaction it has not seen before, IPC registers the server as a *participant* in the transaction. Participants are included in the commit protocol, and have the right to themselves issue requests on behalf of the transaction. Participation spans the interval between receiving a request made on behalf of the transaction and responding to a **vote** request (see Section 4).

3.3 Distributed Transactions

A transaction becomes *distributed* when a request tagged with its *Tid* is issued to a remote node. When a process (client or server) at node A issues a request to a server at node B (S_B), IPC registers TM at node B (TM_B) as a *subordinate* of TM_A, and (as above) registers S_B as a participant with TM_B. Thus, the TM at each node coordinates the local activities of a transaction and the activities of subordinate TMs, and cooperates with superior TMs. The topology of a transaction can be thought of as a directed graph, with the coordinator at the root, TMs at the internal nodes, the owner and servers at the leaves, and arcs pointing in the direction of the superior-subordinate relation (see Figure 3).[4] There is no global knowledge of the transaction topology; each TM only knows its immediate superiors and subordinates. IPC assures that the graph is built with the invariant property that there is always a path connecting the coordinator to each node in the graph. This property is used to assure proper sequencing of operations during commit processing.

Using the graph topology rather than a single centralized coordinator was done for efficiency. The number of network messages is reduced both on the local

[4] This organization is similar to the hierarchy described in [23], with the exception that a TM can have multiple superiors and the graph can have cycles.

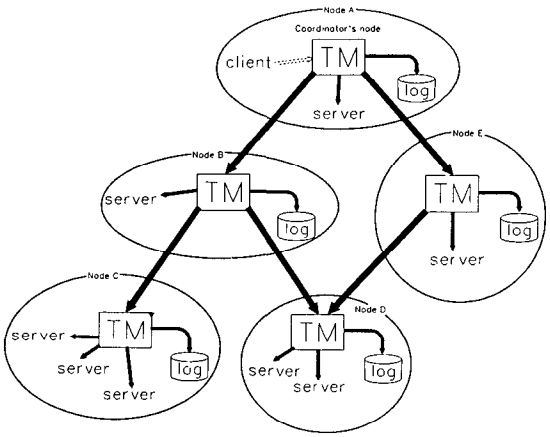

Fig. 3. Structure of a distributed transaction.

network (since servers communicate only with their node's TM using local IPC) and on the internet. For example, QuickSilver's distributed file system is structured such that a file's directory entry and data may reside on different nodes on the same local-area net. When a client accesses a file over the internet, the coordinator only communicates with one of the TMs (e.g., the directory manager's) over the internet; that TM then communicates with the other (e.g., the data manager's) over the LAN. This requires fewer internet messages than would be the case if all TMs communicated directly with the coordinator.

3.4 Transaction Termination and Failure

A TM *terminates* a transaction in response to one of the following conditions:

(1) The transaction's owner calls **Commit** or **Abort**.
(2) The owner process terminates. Normal termination is equivalent to **Commit**, and abnormal termination is equivalent to **Abort**.
(3) A participant calls **Abort**.
(4) A volatile or recoverable participant fails (i.e., terminates before voting).
(5) The local CM detects a permanent connection failure to a node whose TM is a superior in the transaction.
(6) A subordinate TM reports the termination of the transaction.

Any of these conditions cause the TM to initiate its commit processing.

A transaction can *fail* before it terminates. A failed transaction is not immediately terminated; instead, the failure is remembered and the transaction is aborted when it does terminate. This allows nonrecoverable operations (e.g., error reporting) to continue, but ensures that any further recoverable operations will be undone. A TM causes a transaction to fail under any of the following conditions:

(1) A volatile or recoverable participant fails (i.e., terminates before voting).

(2) The local CM detects a permanent connection failure to a node whose TM is a subordinate in the transaction.
(3) A subordinate TM reports the failure of the transaction.

Note that participant failure can cause either transaction failure or termination. Servers declare this when they offer their service. The asymmetry in failure of superior vs. subordinate nodes allows early reclamation of resources for the subordinate, while allowing the error to be seen and reported by the superior.

3.5 Transaction Checkpoints

The **Checkpoint** call allows the owner to save the partial results of a transaction. All the changes to state before the transaction checkpoint take effect permanently and, if the transaction later aborts, it will roll back only to the checkpoint. Servers retain any locks that have been acquired by the transaction, so consistency is maintained across checkpoints. Transaction checkpoints provide a means for long-running applications to survive system crashes and for distributed programs to synchronize themselves without the overhead of starting a new transaction at every synchronization point.

4. COMMIT PROCESSING

QuickSilver commit processing closely follows the distributed commit paradigm described in [23]. However, we will give a brief overview here to establish some terminology that will be used later. A TM initiates commit processing in response to a transaction termination condition.[5] To abort a transaction, a TM sends **abort** requests to all of its participants and immediate subordinate TMs; the latter recursively propagate the **abort** request to their participants and subordinates. In attempting to commit a transaction, the coordinator sends **vote** requests to all of its participants and immediate subordinate TMs; again, the latter recursively propagate the **vote** request. When a recipient of the **vote** request is *prepared* (recoverably ready to commit or abort), it returns its vote (**vote-commit** or **vote-abort**) by completing the request. To become prepared, a TM must receive **vote-commit** responses to each of its **vote** requests. When the coordinator is prepared, it commits the transaction and sends out **end** requests, which contain the outcome of the transaction (**end-commit**, **end-abort**) and get propagated in a manner similar to the **vote** requests. When all **end** requests have been completed, signifying that all involved parties know that the transaction has committed, the coordinator ends the transaction.

If a participant fails while prepared, it contacts its local TM after restarting to find out the status of the transaction. If a TM fails while prepared, it contacts its superior or the coordinator, whose identity is sent out with the **vote** requests and logged in the **prepare** record.

4.1 Basic Commit Protocols

QuickSilver supports two basic models for committing a transaction; **one-phase** and **two-phase** commit. Servers declare which protocol they follow when they offer their service.

[5] Note that only the coordinator can initiate a commit, but any subordinate TM can initiate an abort.

The one-phase protocol is used by servers that maintain only volatile state. TM sends an **end** request to each one-phase participant to notify it that the transaction is terminating. QuickSilver supports three variants of the one-phase protocol, allowing the server to be notified at different points in the commit processing. These are:

(1) **One-phase immediate.** The server is notified during the first (vote) phase of the two-phase protocol. An example is the window manager, which can reclaim its resources (windows) without waiting for commit processing to complete.
(2) **One-phase standard.** The server is notified when the transaction commits or aborts. Most servers use this variant.
(3) **One-phase delayed.** The server is notified when the commit processing has ended. An example server is CM, which cannot clean up its state (e.g., virtual circuits) until after the commit protocol has completed.

The two-phase protocol provides both synchronization and recoverability. The protocol used by QuickSilver is derived from the two-phase presumed-abort protocol described in [23]. Presumed abort has advantages that are important to QuickSilver servers, including its reduced cost for read-only transactions and the ability to forget a transaction after it ends. QuickSilver extends this protocol to distinguish between the synchronization and the recoverability it provides, and to accommodate the directed graph transaction topology. These extensions are discussed below.

4.2 Voting

QuickSilver defines four votes that a participant may return in response to a **vote** request:

(1) **Vote-abort.** The participant forces the transaction to be aborted. It may immediately undo its own actions and is not included in phase two of the commit protocol. The second phase is used to announce the abort to all other participants.
(2) **Vote-commit-read-only.** The participant votes to commit the transaction, declares that it has not modified any recoverable resources, and requests not to be included in phase two of the commit protocol.
(3) **Vote-commit-volatile.** The participant votes to commit the transaction, declares that it has not modified any recoverable resources, but requests to be informed of the outcome of the transaction during phase two.
(4) **Vote-commit-recoverable.** The participant votes to commit the transaction, declares that it *has* modified its recoverable state, and thus requests to be informed of the outcome of the transaction during phase two.

Vote-commit-volatile is an extension of the standard presumed-abort protocol of [23] that allows TM to provide less expensive synchronization for non-recoverable servers, such as those maintaining replicated volatile state, by

minimizing log activity. If no participants or subordinates respond **vote-commit-recoverable**, TM does not write any commit protocol log records.

4.3 Advanced Commit Protocols

QuickSilver guarantees atomicity will be preserved in case of process or machine failure, and even in the case of an improperly functioning client. In part, this is achieved by IPC, which guarantees delivery and ordering of requests, enforces that requests are issued on behalf of valid transactions by valid owners and/or participants, and keeps track of server participation to ensure that the transaction graph is properly connected. However, IPC does not guarantee ordering of requests outside of a single client-server conversation. Since transactions may involve several separate conversations between clients, servers, and TMs, it is still possible for the graph not to be fully formed and stable during commit processing. It is necessary that the commit protocol take this into account. This section discusses some of these problems and their solutions.

4.3.1 *Commit before Participate.* Consider the case where a client commits a transaction before all IPC requests made on its behalf are completed.[6] For example, suppose a client on node A calls a local server (S_{A_1}) and commits without waiting for S_{A_1} to complete the request. Furthermore, suppose S_{A_1} calls S_{A_2} as part of processing the client request. TM may see S_{A_2}'s participation after it has committed the transaction. In such a case, TM would tell S_{A_2} to abort (cf., due to presumed abort) even though the transaction committed. The simple expedient of forbidding the client to call **Commit** with uncompleted requests is not acceptable, since this is a normal state of affairs (e.g., requests for user input).

To ensure that all servers involved in a transaction participate in the commit protocol, TM, the kernel, and servers obey the following rules:

Rule 1. TM must accept new participants and include them in the voting until it commits.

Rule 2. Requests are partitioned into those that must complete before the transaction commits, and those (called "ω-requests") that need not complete because they do nothing that could force the transaction to abort.[7] TM at a node will not decide to commit a transaction until all non-ω-requests issued on the transaction's behalf on that node have completed.

Rule 3. A one-phase server that makes a non-ω-request on behalf of a client transaction (e.g., as part of servicing a request made to it) must make that request before completing the request it is servicing.

These rules are sufficient to ensure that TM will properly include all participants in the commit protocol. One-phase-standard and one-phase-delayed servers

[6] This can occur during a **Commit** by a single-process client with uncompleted asynchronous IPC requests, or by a multiprocess client with uncompleted synchronous requests.

[7] ω-requests include stateless requests (timeouts, polls), requests for user or device input, and the like. All requests to stateless servers (see Section 3.2) are ω-requests. Servers define their interface to allow ω-requests to be identified. Since requests are typed, this is implemented by defining special types for such requests.

receive their **end** message (terminating their participation) after the transaction commits. Rules 2 and 3 ensure that these servers do not issue any requests that could otherwise force the transaction to abort, after it has already committed.

4.3.2 *Cycles in the Transaction Graph.* It is possible for cycles to occur in the transaction graph, for example, when a server on node A (S_A) sends a request on behalf of transaction T to a server on node B (S_B), which itself requests S_C on node C, which requests S_B. In this case, TM_B has two superior, TM_A and TM_C. To accommodate this, each subordinate TM in a transaction distinguishes which of its superior TMs was the first to issue a request since the start of the transaction.[8] The subordinate TM initiates commit processing when this *first superior TM* sends a **vote** request. TM responds **vote-commit-read-only** to the **vote** requests of all other superior TMs, including new ones that appear during commit processing. When TM has collected votes from its local servers and all subordinates, it completes the first superior's **vote** request with its vote on the transaction.

In the above example, TM_A would send a **vote** request to TM_B, which would send a **vote** request to TM_C, which would send a **vote** request to TM_B. TM_B would respond **vote-commit-read-only** to TM_C, which would then (assuming the transaction was going to commit) respond **vote-commit** to TM_B, which would itself respond **vote-commit** to TM_A.

4.3.3 *New Requests after Becoming Prepared.* It is possible for new requests to arrive at a server after it has voted to commit (e.g., if server S_A calls already prepared server S_B). S_B can avoid atomicity problems in a rather heavy-handed way by refusing to process further requests (i.e., returning a bad return code), causing S_A to abort the transaction (S_A can not have voted). However, such is not our way. Instead, a prepared server that receives new work on behalf of a transaction is treated as a new participant. By Rule 1, TM allows the server to **re-vote** by sending another **vote** request to the server, which again becomes prepared and responds **vote-commit**. Here, if S_A and S_B are on different nodes, and if TM_B is already prepared, TM_A becomes the new "first" superior, and TM_B sends a **vote** request to S_B when it receives a **vote** request from TM_A.

It is possible that either TM_B or S_B will not be able to again become prepared, forcing it to respond **vote-abort**. The apparent violation of atomicity is resolved by the observation that the coordinator will not yet have decided to commit and will eventually see the **vote-abort**.

4.3.4 *Reappearance of a Forgotten Transaction.* Some systems [20] allow a node to unilaterally abort a transaction locally without informing any other nodes. If a transaction returns to a node that had locally aborted it, the transaction may be interpreted as a new one and subsequently committed. This will break atomicity, as some of the effects of the transaction will be permanent while some have evaporated with the local abort. The protocol described in [20] uses a system of time-stamps and low-water marks to preserve atomicity in such situations.

[8] Or since the most recent **vote** request (see Section 4.3.3).

In QuickSilver, a TM at a node can unilaterally abort a transaction and forget about it after informing its first superior TM. The effects of the transaction may be rolled back immediately by participants at or beneath that node. It is not necessary to force an abort record to the log, since any node failure prior to completion will cause the transaction to abort anyway. Our approach requires remembering the aborted transaction until the parent knows about its aborted status, but saves the extra bookkeeping of time-stamps and low-water marks associated with all work requests required by the protocol described in [20].

4.4 Coordinator Reliability

The coordinating TM is ordinarily the one at the transaction's birth-site. In the performance-critical case of a strictly local transaction, this is the correct choice. Most transactions are created by user workstations, which are the most likely to fail (e.g., when the user bumps the power switch or turns off the machine to go home). Coordinator failure during execution of the commit protocol for a transaction involving resources at remote recoverable servers can cause resources to be locked indefinitely.

QuickSilver uses two mechanisms to harden the coordinator. Both solutions—coordinator migration and coordinator replication—are cheaper and simpler than the Byzantine agreement protocol proposed by other researchers [24].

4.4.1 *Coordinator Migration.* At commit time, when the coordinator knows that a transaction has become distributed, it can designate a subordinate TM to take over as the coordinator. The topology of the transaction is changed to reflect the fact that the birth-site TM becomes a subordinate. Migration is used when the coordinator has only a single subordinate, in which case the subordinate is selected as the new coordinator. This corresponds to the common case of a program accessing files at a remote file server. Migration is accomplished by the coordinator (TM_A) first requesting votes from its local servers. After they respond, TM_A sends a special variant of the **vote** request to the subordinate (TM_B), naming it as the new coordinator, and specifying if TM_A needs to be included in phase two of the commit protocol. TM_B takes over the role of coordinator, requesting votes from its participating servers and subordinate TMs.

Migration tends to locate the coordinator where the transaction's shared, recoverable resources are (e.g., at a file server), which reduces the probability of a functioning server having to wait for a failed coordinator. When, as is often the case, TM_A has no two-phase participants in the transaction, coordinator migration also saves a remote IPC request. However, the migrated coordinator is still a single point of failure.

4.4.2 *Coordinator Replication.* For transactions in which the coordinator has multiple subordinates, QuickSilver allows the coordinator to be replicated to shorten the interval during which it is vulnerable to single-point failures. The basic idea is to select a subset of the subordinates as backup coordinators, to use a hierarchical two-phase commit protocol between the remainder of the subordinates and the coordinators, and to use a special protocol among the coordinators. In theory, one can use any number of replicas and any suitable protocol

(e.g., Byzantine agreement) among the replicas. QuickSilver uses a simple two-way replication algorithm.

A coordinator TM_A replicates itself by sending a special variant of the **vote** request to a subordinate TM_B. TM_B then sends a **vote** request to TM_A. This partitions the transaction graph into two blocks, one composed of coordinator TM_B and its subordinates, and one composed of coordinator TM_A and all its other subordinates. TM_A and TM_B regard each other as their standby. TM_A and TM_B then send **vote** requests to subordinates in their respective blocks, including in the request the name of both coordinators. The following describes the protocol from TM_A's standpoint; TM_B behaves likewise. When all TM_A's participants and subordinates have responded **vote-commit**, TM_A forces a **prepared** log record and then completes TM_B's **vote** request. When it is prepared *and* it has received the completion of its **vote** request to TM_B, it sends an **end-commit** request to TM_B. Upon receiving an **end-commit** request from TM_B, TM_A forces its **commit** record and sends **end-commit** requests to its subordinates. When these requests have been completed, TM_A completes TM_B's **end-commit** request. When TM_B has completed TM_A's **end-commit** request, TM_A writes its **end** record. If TM_B fails, then TM_A aborts if it has not yet sent **end-commit** to TM_B, otherwise it remains prepared. If a coordinator fails, its subordinates contact the standby to determine the outcome of the transaction.

The protocol blocks only if a failure occurs during the exchange of **ready** messages (an exceedingly short interval in practice). The cost is the **vote** and **end** requests from TM_B to TM_A, the **prepared** log record at TM_A, and the **commit** record at TM_B.

5. LOGGING AND RECOVERY

Each node's recovery manager contains a Log Manager (LM) component that is used by TM to write its commit protocol log records and is also used by other servers[9] that manage recoverable data. Providing a common log for use by all servers imposes conflicting goals of generality and efficiency on LM. If one were to port a significant subsystem, such as a database manager, to QuickSilver, LM is intended to be general enough to not force restructuring of the database's recovery architecture, and efficient enough to allow the database to run without significant performance penalty.

Of these two goals, efficiency was the simpler to achieve. For example, because of the use of a common log, servers can take advantage of TM's log forces to avoid doing their own during commit processing. Generality is more difficult, as even a single database manager or file system contains many storage structures, with different recovery techniques being most appropriate for each. Rather than trying to impose a fixed set of recovery techniques on such servers, LM offers a relatively low-level interface to an append-only log. This interface provides a core set of services, including restart analysis, efficient access methods, and archiving. On top of this interface, servers implement their own recovery

[9] The log may in fact be used by any recoverable program (e.g., long-running applications), but to simplify the text we will call any program that calls LM a "server".

algorithms and, in fact, drive their own recovery. This allows them to tailor their recovery techniques to those most appropriate for the data they maintain.

5.1 Log Manager Interface

The log consists of a large, contiguous address space subdivided into 512-byte *log blocks.* Each byte is addressable by a unique, 64-bit *log sequence number* (LSN). The log is formatted into *log records.* Each log record contains an abbreviated version of the *recovery name* used by servers to identify their log records, the *Tid,* and the server's data. Records may be of any length and can span any number of log blocks. Records from different transactions and different servers are freely intermixed in the log.

Before using the log, servers call LM to **identify** themselves, specifying their recovery name and the optional log services they require (see below). The server can then **read**, **write**, or **force** (synchronously write) records to the log. **Write** and **force** return the record's LSN.

A server can **read** records from the log in one of two ways: by providing the actual LSN or, more commonly, by opening a *scan* on the log via a logical *cursor.* A server can scan all its records, or just those of a particular transaction. LM returns the data, the *Tid,* and the status of the transaction (e.g., Prepared, Committed, Aborted). A server can read only valid records with its recovery name. To locate a starting point for recovery, servers are provided access to the *log restart area.* Servers typically save the LSN of a *log checkpoint* (see below) in the log restart area.

5.2 Log Operation and Services

LM formats log records received from servers into log blocks and buffers them. Buffered blocks are written to the nonvolatile *online log* either when buffer space is exhausted or when a server (or TM) calls **force**. The online log is structured as a circular array of blocks. Newly written blocks are written as the *head* of the log; each newly written log block overlays the oldest previously existing block. If that block still contains *live* data (i.e., data that is still needed by some server), it is copied to a log *archive* from which it can still be read, although perhaps at a performance penalty. Because the log is common to all servers, **force** causes all previously written records from all servers to be written to nonvolatile storage. TM exploits the common log to reduce the number of log forces during commit processing. When a server responds to TM's **vote** request, it specifies an LSN that must be forced before the server can enter the prepared state. Thus the log needs to be forced only once per transaction (by TM), regardless of the number of servers writing log records.

TM and each server have a distinct *log tail,* which is the oldest record they will need for crash recovery. When LM's newly written log records approach a server's log tail, LM asks the server at its option to take a *log checkpoint.*[10] In response, a server performs whatever actions are necessary (e.g., flushing buffers or copying log records) to move its log tail forward in the log and thus avoid having to access archived data during recovery.

[10] Log checkpoints are distinct from transaction checkpoints, described in Section 3.5.

When servers identify themselves to LM they define which optional log services they require. Dependencies between options are minimized so that, where possible, servers are not penalized for log services they do not use, nor by log services used by other servers. LM provides the following optional log services:

(1) **Backpointers on log records.** Servers that modify data in place need to replay their records for aborted transactions. This can be done efficiently by requesting LM to maintain backpointers on all log records written by that server for each transaction.

(2) **Block I/O access.** Servers that write large amounts of log data can call a set of library routines that allow them to preassign a contiguous range of log blocks, construct their own log records in these blocks, and write them as a unit, rather than calling LM for individual records. Since servers and LM are in separate address spaces, servers must explicitly write their preassigned blocks (i.e., they are not automatically written by TM log forces). Crashes can therefore create "holes" in the *physical* log. LM bounds the maximum contiguous range of preassigned blocks, and thus can recognize holes and skip over them during recovery. The holes will not affect the *logical* log of the client, which will always be contiguous.

(3) **Replicated logs.** Servers managing essential data may require the log to be replicated to guard against log media failure. LM supports log replication either locally or to remote disk servers.

(4) **Archived data.** Log blocks are archived when the online log wraps around on live data. This includes log records from inoperative servers (i.e., those that have crashed and not yet restarted), records from servers that do not support log checkpoints, and certain other records (e.g., records necessary for media recovery, records of long-running transactions). Except for a performance difference, the fact that a record is archived rather than in the online log is transparent to a server reading the record. The archive is stored in a compressed format so that only records containing live data are stored. The archive may be replicated, either locally or to remote archive servers.

5.3 Recovery

During recovery, LM scans the log starting with TM's log tail. This *analysis* pass determines the status (prepared, committed, aborted) of each transaction known to TM at the time of the crash, and builds pointers to the oldest and newest records for each such transaction written by each server. It also builds an *index*, used to support scans, that maps each server's log records to the blocks that contain them.

At the completion of the analysis pass, LM starts accepting *identify* requests from servers. At this point, servers begin their own recovery. In QuickSilver, servers drive their own recovery by scanning the log and/or by randomly addressing log records. Scans may retrieve all records written by the server or only the records written by a specific transaction. The server determines the number of passes over the log and the starting position and direction of each pass, and implements the recovery actions associated with the log records.

By remaining independent of the recovery protocol and by allowing servers to drive their own recovery, the QuickSilver recovery manager potentially incurs a higher cost during recovery than if it restricted the recovery protocols used by clients and drove recovery from the contents of the log. We attempt to minimize this cost in several ways. The index over the log maintained by LM allows it to read only the blocks that actually contain a server's data. The index, in association with the directional information associated with log scans, allows LM to prefetch data blocks in anticipation of the client's **read** requests. Also, the LM-maintained backpointers minimize the cost of backward scans. Finally, the results of the LM analysis pass are made available to servers. For some three pass-recovery protocols [14, 15, 33] the results of the TM's recovery can be used to simplify, or even to replace, the first pass of the protocol.

6. PERFORMANCE ANALYSIS

Table I summarizes the costs incurred by the QuickSilver recovery manager. There is one column for each of the four commit protocols, *One-Phase*, *Read Only*, *Two-Phase Volatile*, and *Two-Phase Recoverable*. The *Cost per Transaction* is a fixed overhead that is incurred regardless of the number of participants or the distribution. For rows in this section, and in the *Cost per Subordinate* section, the protocol column is selected as the maximum of that node's participant protocols and its subordinate protocols. The *Cost per Subordinate* rows show IPC requests between TMs on different nodes, and commit protocol log writes at the subordinate node. The protocols used between TMs are always two-phase.[11]

To allow comparison with other systems running on other hardware, Table II shows the cost of the base operating system functions used by the recovery manager. These (and all later benchmarks) were measured on RT-PC Model 25s (about 2 RISC mips) with 4 megabytes of memory, IBM RT-PC token ring adapters (4 megabit/sec transfer rate), and M70 70 megabyte disks (5 megabit/sec transfer rate). QuickSilver, as well as all server and client processes, were compiled with the PL.8 compiler [2]. TM uses short IPC messages, and LM uses streamed writes. The table entries for 1K-byte IPC messages and random-access disk I/O will be used in later benchmarks. Remote IPC performance was measured on a lightly loaded network, but because of the characteristics of the token ring, performance does not degrade significantly until network load exceeds 50 percent.

Given these base system and I/O performance numbers, a series of benchmarks was run to determine the actual overhead of the transaction and logging mechanism. We used a set of benchmarks similar to those reported for Camelot in [36], which, in addition to providing a way of determining the recovery management overhead, allows a direct comparison of QuickSilver's performance to that of at least one other system under a similar set of conditions.

All benchmarks were run on otherwise unloaded machines, with an unloaded network, a simplex log, and unreplicated coordinators. Each number in Table III

[11] Table I describes the case where the coordinator is not being replicated. The cost per transaction increases by two remote IPC requests and two force log writes when the coordinator is replicated (see Section 4.4.2).

Table I. Transaction Management Algorithmic Costs

	Transaction Protocol			
Cost per transaction	One phase	Read only	Two-phase volatile	Two-phase recoverable
Begin transaction	1 Local IPC	1 Local IPC	1 Local IPC	1 Local IPC
Commit/abort transaction	1 Local IPC	1 Local IPC	1 Local IPC	1 Local IPC
Log commit/abort record	0	0	0	1 Log force
Log end record	0	0	0	1 Log write
Cost per participant				
Request vote	0	1 Local IPC	1 Local IPC	1 Local IPC
Commit/abort transaction	1 Local IPC	0	1 Local IPC	1 Local IPC
Cost per subordinate				
Request vote	—	1 Remote IPC	1 Remote IPC	1 Remote IPC
Commit/abort transaction	—	0	1 Remote IPC	1 Remote IPC
Log prepare record	—	0	0	1 Log force
Log commit/abort record	—	0	0	1 Log force
Log end record	—	0	0	1 Log write

Table II. Primitive Operation Times in msecs.

Primitive operation	Time
Local 32-byte IPC	.66
Local 1K-byte IPC	1.16
Remote 32-byte IPC	9.0
Remote 1K-byte IPC	16.0
Average 512-byte streamed raw disk I/O, including cylinder steps	2.3
Random-access 4096 byte I/O, read or write	37.5

is the per-transaction average over 4 runs, each run consisting of a batch of 4096 32-byte transactions or 512 1K-byte transactions. The write benchmarks caused log checkpoints, and the time for these are included in the averages. As in [36], the benchmark transactions were run serially from a single application, and all service requests were synchronous, as the goal was to measure transaction management overhead as opposed to response time or throughput.

The following benchmarks were run:

(1) Transactions on 1, 2, and 3 local servers that read or write one 32-byte record. These demonstrate the basic overhead of local read and write transactions, and the incremental cost of involving additional servers.

(2) Transactions on 1, 2, and 3 local servers that read or write ten 32-byte records (as ten separate synchronous requests). These allow computing the incremental costs of additional operations on servers from an existing transaction, which then allows computing the local per-transaction overhead, including that of log forces for write transactions.

Table III. Benchmarks on 1–4 RT-PCs, msecs/transaction

Transaction benchmarks	1 Server	2 Servers	3 Servers
Local reads			
1 32-byte read/server	6.1	8.7	11.2
10 32-byte reads/server	14.1	24.7	35.2
1 1K-byte read/server	6.6	9.6	12.6
10 1K-byte reads/server	18.7	33.8	48.9
Local writes			
1 32-byte write/server	41.7	41.7	42.4
10 32-byte writes/server	58.4	92	125
1 1K-byte write/server	41.7	50.8	66.9
10 1K-byte writes/server	119	181	239
Remote reads			
1 32-byte read/server	31.9	45.5	58.8
10 32-byte reads/server	121	224	329
1 1K-byte read/server	38.1	57.9	77.5
10 1K-byte reads/server	183	348	515
Remote writes			
1 32-byte write/server	77	101	122
10 32-byte writes/server	201	325	447
1 1K-byte write/server	80	124	152
10 1K-byte writes/server	335	533	725

(3) All of the above with 1K-byte records.

(4) All of the above with the application on one node and each data server on a separate node. This demonstrates the additional overhead for distributed server requests and committing distributed transactions.

The numbers in Table III were used to derive the transaction management costs shown in Table IV. For example, for local read-only transactions there is a fixed overhead of 3.5 msec. and a per-server overhead of 1.7 msec. Comparing these numbers with the numbers that can be derived from the primitive operation times from Table II and the algorithmic operation costs from Table I allows one to get a rough idea of the execution time of TM and LM. For example, simple read transactions require two IPC requests (**Begin** and **Commit**) plus one IPC request (**Vote**) for each of the n participating servers. This adds up to $1.32 + .66n$, so the execution time in TM is approximately $2.18 + 1.04n$ msec.

The equations for read transactions in Table IV closely match the benchmark data points. The write transactions were more difficult to measure accurately. Benchmark transactions were run serially, and were the only transactions running in the system. Because LM physically writes its log contiguously on the disk, a complete revolution is missed between transactions, and transaction execution is effectively synchronized to the disk rotation rate. Transactions arriving at random times would see faster response time. It is also interesting to note that if other log activity were occurring that allowed the log to "keep up" with disk rotation, response times for small transactions could be much lower than those observed in the benchmarks. To allow the CPU overhead of write transactions to be observed independently of the effects of disk rotation, Table V shows the results of repeating the local write benchmark with the LM disk driver call changed from "write" to "no-op".

Table IV. Approximate Elapsed Times in msecs of Various QuickSilver Functions on the RT-PC

QuickSilver function	Time
Cost/transaction for n servers	
Local read-only	$3.5 + 1.7n$
Local write	$31.0 + 4.2n$
Remote read-only	$18.5 + 3.6n$
Remote write	$53.0 + 7.3n$
Cost/read operation	
Local 32-byte	0.89
Local 1K-byte	1.35
Remote 32-byte	9.9
Remote 1K-byte	16.1
Cost/write operation	
Local 32-byte	2.8
Local 1K-byte	7.2
Remote 32-byte	13.8
Remote 1K-byte	28.3

Table V. Local Write Transaction CPU Cost

Transaction benchmarks	1 Server	2 Servers	3 Servers
1 32-byte write/server	19.5	25.7	31.7
10 32-byte writes/server	43.1	72.7	102
1 1K-byte write/server	20.5	28.4	35.4
10 1K-byte writes/server	57.4	103	148

QuickSilver function	Time
Cost/transaction for n servers	
Local write	$13.2 + 4.2n$
Cost/write operation	
Local 32-byte	2.6
Local 1K-byte	4.1

Table VI. Remote Read Transactions, 1 32-Byte Asynchronous Read/Server, msecs/transaction

Number of servers	Time (change vs. synch)
1 Server	32.0 (+0.3%)
2 Servers	37.5 (−17.6%)
3 Servers	42.9 (−27.0%)

Finally, it is important to point out that the raw performance numbers are intended to be used to derive the per-transaction operational costs. They do not exploit possible parallelism, and thus are not an indication of potential throughput. To illustrate this, we repeated the benchmark for 32-byte remote read transactions, but changed it to make asynchronous requests to the servers. The results, with percent changes from Table III, are shown in Table VI. The slight decrease in time for one node is due to the extra kernel calls to wait on the group of requests. The fact that execution time grows with the number of servers shows that parallelism is not perfect; all messages go to or from the client's node, so the network and the client node's Communication Manager act as a bottleneck.

7. RELATED WORK

Several systems described in the literature use transaction-based recovery as a lower-level component of a higher-level entity such as a file system or database. For example, System R and R* [15, 19] implement relational databases that support atomic transactions. Locus [26, 38] offers a transactional file system.

Other systems, such as Argus [21], Eden [30], Clouds [1], CPR [8], and Avalon [16] implement programming languages that include constructs for recoverable data objects built on top of a lower-level transaction-based recoverable storage manager.

Camelot [36] (and its precursor TABS [34]) integrates transaction-based recovery and write-ahead logging with virtual memory in a manner similar to CPR, but uses software rather than special-purpose hardware to control access to recoverable storage. Camelot offers recoverable storage in the context of a standard programming language (C) via a macro package and library routines. Camelot macros hide logging, recovery, and commit processing from servers that manage recoverable resources. Applications start and end transactions and call servers via Camelot macros that generate Mach RPC calls. Unlike Argus and CPR (which support redo logging), Camelot implements both redo and undo/redo logging. It also allows a choice of blocking or nonblocking commit protocols, and supports nested transactions [25] in a manner similar to Argus.

The V-System [9] implements transactions on top of its process group facility [10, 11]. In V, transactions are implemented via a transaction library running as part of the client process, a transaction log server, and data servers that manage recoverable objects. Each transaction is represented by a process group. A client calls the (possibly replicated) log server to create a transaction, adds each transactional server it calls to the transaction's process group, and passes the transaction ID as a parameter to the server. The client multicasts **prepare-to-commit** messages to the group, and when all respond affirmatively, calls the log manager to commit the transaction.

While there is much in QuickSilver recovery management that is similar to the aforementioned systems, QuickSilver differs from them in several significant ways. The basic difference is the use of transactions as a unified recovery mechanism for both volatile and recoverable resources. This motivated the lightweight extensions to the commit protocol and is reflected in the low overhead exhibited in the benchmarks. This also led to the fact that QuickSilver directly exposes the recovery management primitives at a lower level than most comparable systems.[12] In particular, servers implement their own recoverable storage, choose their own log recovery algorithms, and drive their own log recovery. In addition to being more flexible and potentially more efficient for servers developed especially for QuickSilver, this approach also simplifies porting recoverable services developed for other systems to QuickSilver by mapping their existing recovery algorithms onto the corresponding QuickSilver primitives.

Another important difference is QuickSilver's integration of recovery management into IPC. There is no special "server call" mechanism for recoverable

[12] Camelot offers a "primitive interface" that allows servers more direct control of their storage, but encourages using the higher-level library.

servers as in Camelot. This, plus the QuickSilver notion of system-created "default" transactions, allows client programs written in conventional programming languages to be completely unaware of the recovery mechanism and still behave atomically. This greatly facilitates porting programs such as file-processing applications to QuickSilver. IPC automatically tracks server participation in transactions, which eliminates the need for system calls to add servers to transactions as in V [10], and allows implementing the "re-vote" mechanism, which itself eliminates the need for distinct "close" calls to servers to quiesce them prior to initiating commit processing.

8. STATUS AND CONCLUSIONS

QuickSilver is installed and running in daily production use on 47 IBM RT-PC's in the computer science department at IBM Almaden Research Center and at other IBM locations. In addition to the QuickSilver group, several other research projects are using QuickSilver as an environment to develop applications and network-based services. The recovery manager has been implemented and is being used as the recovery mechanism for all QuickSilver servers.

The benchmarks described above showed that the recovery management overhead is small. In the simple, normal case (i.e., the transaction commits, no failures occur, and no re-vote is required), the recovery manager requires a minimal number of messages, and CPU overhead is very small. Experience with the system has confirmed that recovery management overhead is negligible and not perceptible to users. We believe this shows that the mechanism is efficient enough to be used for servers with very stringent performance demands.

We were concerned when we started the design of the recovery manager that there would be exactly two types of users: simple servers, like the window manager that just need a completion notification mechanism; and the file system, which needs distributed two-phase commit in its full glory. In fact, we have found transactions to be useful in a variety of applications, and have found the ability to decouple commit coordination from logging and to use them individually to be valuable as well. We mentioned several such cases: the replicated name server, which uses commit coordination but not logging, and checkpointable applications, which use logging but not the commit protocol. We are experimenting with other applications, including a distributed messaging facility and a mail store-and-forward system.

Development of the QuickSilver Distributed File Services (DFS) [7] confirmed the use of server-defined recovery algorithms and server-driven recovery. The approach taken by other systems of embedding all recovery processing within the recovery manager simplifies programming servers. However, DFS pointed out several shortcomings in this style of transparent recovery. Certain operations on file system metadata require operation logging (e.g., B-tree inserts that may provoke splitting the tree are undone/redone operationally to avoid locking large subtrees). The DFS storage allocator, which uses a bitmap, implements its own value logging and concurrency at the bit level, a granularity not to our knowledge supported by any of the aforementioned recovery managers. Transaction checkpoints were motivated by various DFS metadata updates (e.g., B-tree splits) that are done on behalf of DFS internal transactions rather than client transactions

to maximize concurrency and eliminate unnecessary undo operations after client-transaction aborts. Checkpoints allow the updates to be logged and recovered without the overhead of starting a new transaction for each one. Experience seems to show that the log index, prefetch during scans, and a reasonable amount of log buffer memory provide more than adequate recovery performance.

We intend to pursue development of the QuickSilver recovery manager in the following areas:

(1) **Deadlock Detection.** As mentioned earlier, no work has been done on the Deadlock Detection component of the recovery manager. We anticipate beginning this work shortly.

(2) **High-Performance Servers.** The "block access" log interface reduces the number of calls to the log manager, but causes sparser utilization of log blocks and more log block writes. Considerably more performance analysis is necessary to evaluate the benefit of block access for servers like the file system that potentially log large amounts of data.

(3) **Nested Transactions.** QuickSilver presently does not include a nested transaction mechanism. The utility of a mechanism such as that proposed by Moss [25] is clear, and we intend to investigate implementing one.

(4) **Recoverable Object Managers.** QuickSilver's recovery manager is intended primarily as a tool for use by low-level servers, and as such trades ease of use to gain flexibility and efficiency. However, we recognize the merit of systems like Camelot and Argus that make it easy to define and use recoverable objects. It is relatively straightforward to implement recoverable object managers on top of the QuickSilver recovery primitives. We intend to explore a language-directed facility for defining and using recoverable objects, perhaps in the context of a language such as C++.

ACKNOWLEDGMENTS

We would like to thank Jim Wyllie and Luis-Felipe Cabrera, whose work on the architecture and implementation of the QuickSilver Distributed File Services has helped to drive the design of the recovery manager and has provided a testbed to debug it.

REFERENCES

1. ALLCHIN, J. E., AND MCKENDRY, M. S. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 31–44.
2. AUSLANDER, M., AND HOPKINS, M. An overview of the PL.8 compiler. In *SIGPLAN '82 Symposium on Compiler Writing* (Boston, Mass., June 1982). ACM, New York, 1982.
3. BARON, R. V., RASHID, R. F., SIEGEL, E. H., TEVANIAN, A., JR., AND YOUNG, M. W. MACH-1: A multiprocessor oriented operating system and environment. In *New Computing Environments: Parallel, Vector, and Systolic*, SIAM, 1986, 80–89.
4. BARTLETT, J. A NonStop kernel. In *ACM Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif. Dec. 1981). ACM, New York, 1981, 22–30.
5. BIRMAN, K. P. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 79–86.

6. BORR, A. J. Transaction monitoring in Encompass: Reliable distributed transaction processing. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 1981), IEEE, New York, 1981, 155–165.
7. CABRERA, L. F., AND WYLLIE, J. C. QuickSilver distributed file services: An architecture for horizontal growth. IBM Res. Rep. RJ5578, Feb. 1987.
8. CHANG, A., AND MERGEN, M. F. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst.* This issue, 28–50.
9. CHERITON, D. R. The V kernel: a software base for distributed systems. *IEEE Softw. 1*, 2 (April 1984), 19–42.
10. CHERITON, D. R. Fault-tolerant transaction management in a workstation cluster. Unpublished.
11. CHERITON, D. R., AND ZWAENEPOEL, W. Distributed process groups in the V kernel. *ACM Trans. Comput. Syst. 3*, 2 (May 1985), 77–107.
12. COOPER, E. C. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 63–78.
13. CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. IBM Res. Rep. RJ5244, IBM, San Jose, Calif., July 1986.
14. GRAY, J. N. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller, Eds. Springer-Verlag, New York, 1978, 393–481. Also available as IBM Res. Rep. RJ2188, IBM Almaden Research Center, San Jose, CA 95120.
15. GRAY, J. N., MCJONES, P., BLASGEN, M. W., LORIE, R. A., PRICE, T. G., PUTZOLU, G. F., AND TRAIGER, I. L. The recovery manager of the System R database manager. *Comput. Surv. 13*, 2 (June 1981), 223–242.
16. HERLIHY, M. P., AND WING, J. M. Avalon: Language support for reliable distributed systems. Tech. Rep. CMU-CS-86-167, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., Nov. 1986.
17. INTERNATIONAL BUSINESS MACHINES. Systems Network Architecture Transaction: Programmer's Reference Manual for LU Type 6.2, IBM Corporation GC30-3084.
18. LAMPSON, B. W. Atomic transactions. In *Distributed Systems—Architecture and Implementation.* Springer-Verlag, New York, 1981, 246–264.
19. LINDSAY, B., HAAS, L., MOHAN, C., WILMS, P., AND YOST, R. Computation and communication in R*: A distributed database manager. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 1983). ACM, New York, 1983, 1–10. Also available as IBM Res. Rep. RJ3740, IBM, San Jose, Calif., Jan. 1983.
20. LINDSAY, B. G., SELINGER, P. G., GALTIERI, C., GRAY, J. N., LORIE, R. A., PRICE, T. G., PUTZOLU, F., TRAIGER, I. L., AND WADE, B. W. Single and multi-site recovery facilities. In *Distributed Data Bases*, I. W. Draffan and F. Poole, Eds. Cambridge University Press, Cambridge, UK, 1980. Also available as Notes on Distributed Databases, IBM Res. Rep. RJ2571, IBM, San Jose, Calif., July 1979, 44–50.
21. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 381–404.
22. LYON, B., AND SAGER, G. Overview of the SUN network file system. SUN Microsystems, Inc., Mountain View, Calif., Jan. 1985, 1–8.
23. MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986), 378–396. Also available as IBM Res. Rep. RJ5037, IBM, San Jose, Calif., Feb. 1986.
24. MOHAN, C., STRONG, H. R., AND FINKELSTEIN, S. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 89–103. Also IBM Res. Rep. RJ3882.
25. MOSS, E. B. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, Mass., 1985.
26. MÜLLER, E. T., MOORE, J. D., AND POPEK, G. J. A nested transaction mechanism for LOCUS. In *Proceedings of the 9th ACM Symposium on Operating System Principles* (Bretton Woods, N.H., Oct. 1983). ACM, New York, 1983, 71–89.
27. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2 (June 1982), 187–208.

28. OKI, B., LISKOV, B., AND SCHEIFLER, R. Reliable object storage to support atomic actions. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 147–159.
29. POPEK, G., WALKER, B., CHOW, J., EDWARDS, D., KLINE, C., RUDISIN, G., AND THIEL, G. LOCUS: A network transparent high reliability distributed system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 1981). ACM, New York, 1981, 169–177.
30. PU, C., NOE, J. D., AND PROUDFOOT, A. Regeneration of replicated objects: A technique and its Eden implementation. In *Proceedings of the 2nd International Conference on Data Engineering*, (Los Angeles, Feb. 1986). IEEE Press, New York, 1986, 175–187.
31. RASHID, R., AND ROBERTSON, G. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 1981). ACM, New York, 1981, 64–75.
32. REED, D., AND SVOBODOVA, L. SWALLOW: A distributed data storage system for a local network. In *Networks for Computer Communications*, North-Holland, Amsterdam, 1981, 355–373.
33. SCHWARZ, P. M. Transactions on Typed Objects. Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., Dec. 1984. Available as CMU Tech. Rep. CMU-CS-84-166.
34. SPECTOR, A. Z., BUTCHER, J., DANIELS, D. S., DUCHAMP, D. J., EPPINGER, J. L., FINEMAN, C. E., HEDDAYA, A., AND SCHWARZ, P. M. Support for distributed transactions in the TABS prototype. *IEEE Trans. Softw. Eng. SE-11*, 6 (June 1985), 520–530.
35. SPECTOR, A. Z., DANIELS, D., DUCHAMP, D., EPPINGER, J., AND PAUSCH, R. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985. ACM, New York, 1985, 127–146.
36. SPECTOR, A., ET AL. Camelot: A distributed transaction facility for Mach and the internet—an interim report. Tech. Rep. CMU-CS-87-129, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., June 1987.
37. STONEBRAKER, M. Operating systems support for database management. *Commun. ACM 24*, 7 (July 1981), 412–418.
38. WEINSTEIN, M. J., PAGE, T. W., LIVEZEY, B. K., AND POPEK, G. J. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 115–126.

Received May 1987; revised May 1987; accepted September 1987