Experience with Transactions in QuickSilver

Frank Schmuck
Jim Wyllie

IBM Research Division
Almaden Research Center
Computer Science Department

Abstract

All programs in the QuickSilver distributed system behave atomically with respect to their updates to permanent data. Operating system support for *transactions* provides the framework required to support this, as well as a mechanism that unifies reclamation of resources after failures or normal process termination. This paper evaluates the use of transactions for these purposes in a general purpose operating system and presents some of the lessons learned from our experience with a complete running system based on transactions. Examples of how transactions are used in QuickSilver and measurements of their use demonstrate that the transaction mechanism provides an efficient and powerful means for solving many of the problems introduced by operating system extensibility and distribution.

1 Introduction

QuickSilver is an experimental distributed operating system that was developed at the IBM Almaden Research Center for the IBM RT-PC and IBM RISC System/6000 families of workstations. It contains a small kernel that provides local interprocess communication (IPC), process and thread management, and linkage for installable interrupt handlers. All other system services are implemented by server processes. Clients communicate with servers using network-transparent IPC. QuickSilver runs on a network of about 40 machines at Almaden, and has been used as the primary computing environment for a subset of our local research community for several years.

QuickSilver was designed to make it easy to write sophisticated distributed programs; the interfaces for accessing files, starting processes, or using any other system service are the same regardless of whether the service is local or remote. State associated with such a distributed computation will be split among many servers on different machines. Hence it is vital that the system provide a mechanism for servers to take appropriate actions to release resources, recover client state, etc., when all or parts of a distributed computation end or terminate prematurely due to a software problem, intervention by the user, or when a machine participating in the computation crashes.

The distinguishing characteristic of QuickSilver is that it has borrowed the notion of *transactions* from the database domain, and extended it to serve as the method used for all resource management in the system. In QuickSilver *every* program runs in the context of transactions. The system provides the transaction management infrastructure for servers that support transactional *atomicity* of updates to the state they maintain. In particular, the QuickSilver file system will undo the effects of all operations performed by a distributed computation if any part of the computation fails before successful completion, i.e., before the transaction associated with the computation commits.

In QuickSilver, transactions are used pervasively throughout the system. This made it necessary to extend the traditional notion of a transaction, for example, to accommodate servers that maintain only volatile state. In order for QuickSilver to be a complete system suitable for hosting its own development, it must of necessity import programs from other systems. This implies that existing programs must be able to run in a transactional environment with little or no modification. The purposes of this paper are to describe both qualitatively and quantitatively how transactions are used

in QuickSilver and to assess the successes and failures in our approach of using transactions to unify resource management in a distributed operating system.

The remainder of the paper is organized as follows. We first describe how transaction management is implemented in QuickSilver. Next we show examples of how transactions are used for various purposes in the system, then present measurements of day-to-day transaction usage on our local network. The next section quantifies the cost of transactions. We then present lessons drawn from our experience with QuickSilver, and close with a discussion of related work and our conclusions.

2 Transaction Management in QuickSilver

In database terminology, a transaction is a collection of operations that represent a unit of consistency and recovery. Transactions provide failure atomicity, recoverability, and isolation [8]. In QuickSilver, transactions are used to manage volatile resources as well as non-volatile, so these properties are not applicable in the same way. In QuickSilver a transaction encapsulates a (possibly distributed) unit of work that may provide failure atomicity, recoverability, isolation, or any subset of these.

Transaction management in QuickSilver consists of a toolkit that allows clients and servers to choose among several services related to providing the traditional properties of transactions. Although the QuickSilver system enforces that all computation is done in the context of some transaction, the exact semantics attached to a transaction may vary among different servers, depending on choices made by the implementer of a service. The QuickSilver file system, for example, ensures that updates made by a transaction are atomic with respect to failures, and are permanent once the transaction commits, but it provides a degree of consistency that falls short of full serializability[1].

The QuickSilver transaction management toolkit consists of several pieces:

- transactional IPC,
- a transaction manager, and
- a log manager.

We will briefly describe each component of the toolkit in the remainder of this section. An earlier paper [10] presents the overall architecture of transaction management in QuickSilver and describes the toolkit in more detail.

Transactional IPC. IPC in QuickSilver follows the request-response paradigm. It is similar to IPC in the V system [4], except that QuickSilver IPC requests may be made asynchronously. A unique feature of QuickSilver IPC is that it is transactional; that is, all interprocess communication must be done on behalf of some transaction. There is no escape from this aspect of transactions in QuickSilver. Every IPC request carries with it a *transaction id* (TID) identifying the transaction that made the request. The kernel enforces the restriction that only processes *participating* in a transaction may make requests on its behalf. The process that creates a transaction is a participant, as are processes that have received an IPC request containing its TID.[2] Thus, a server may call upon other servers in order to fulfill client requests as part of a transaction.

When a process participating in a transaction makes an IPC request to a remote server, the kernel routes the request to the local *communication manager* process (CM). The local CM sends the request to its peer on the destination machine. The two CM processes cooperate to handle the details of network transport and recovery from intermittent communication errors. The remote CM registers the transaction with its local kernel and forwards the request to the destination server using local IPC. The IPC response flows back along the same path. (See Figure 1.)

Transaction Manager. The QuickSilver *transaction manager* (TM) is a server process that handles the initiation and termination of transactions. Transactions are created by an IPC request to TM. TM assigns a globally unique TID and registers it with the kernel. The process that created a transaction may call `commit`, and any participant in the transaction may call `abort`. Servers that support recoverable state must undo changes to their state upon transaction abort, or make those changes permanent at commit. The primary purpose of TM is to coordinate the decision to commit or abort among all of the (potentially distributed) participants in a transaction.

When a server process offers an IPC service, it specifies a *participation class*, which determines the protocol used by TM to contact the server at the conclusion of transactions that have called that service. The purpose of having several participation classes is to accommodate the varying demands servers place on the transaction mechanism. The class *no-state* is intended

[1] Following the terminology introduced by Gray et al. [6], the QuickSilver file system provides degree 2 consistency for file updates and degree 1 consistency on directories.

[2] The actual kernel protocol is somewhat more complex [10].

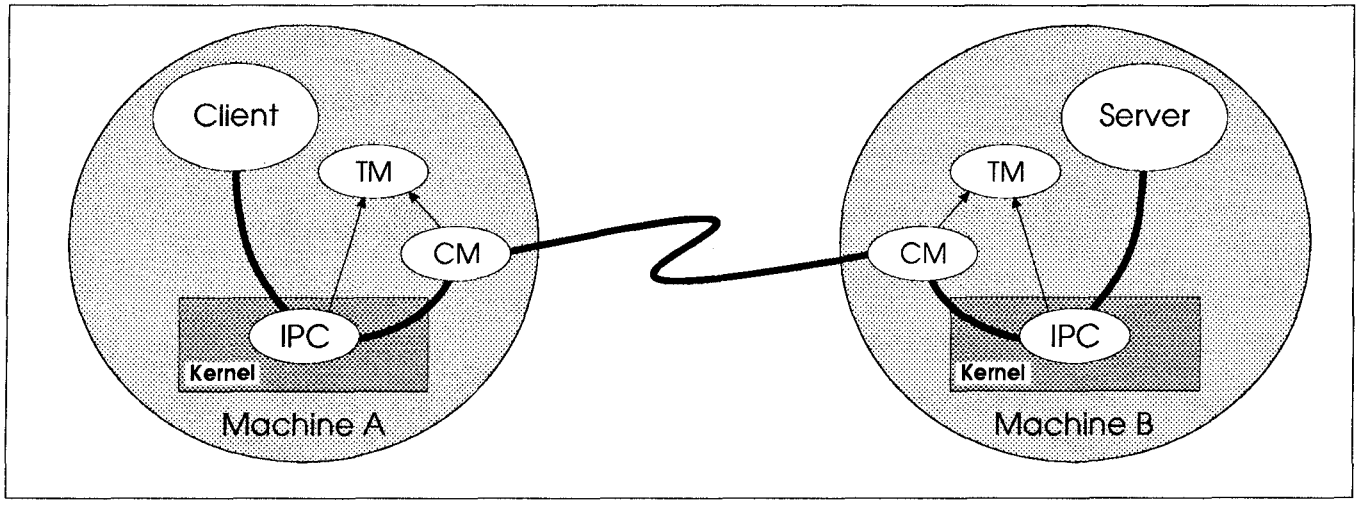

Figure 1: Remote IPC in QuickSilver

for stateless servers that require no notification at all of transaction termination. Some servers require only a single notification that a transaction has ended, to clean up volatile state they hold. There are several varieties of such *one-phase* participation classes, which differ in the point during commit processing at which servers receive their notification. Other servers, primarily those maintaining recoverable state, require a full *two-phase* commit protocol [8, 14]. Examples of servers and their participation classes are given in the next section.

When a process calls `commit` or `abort`, TM collects the IPC participation information from the kernel, then notifies each participating local server according to its participation class. If CM is a participant, TM asks CM for the list of machines to which it has sent IPC requests on behalf of the ending transaction. For each such machine, TM requests that its remote peer recursively perform local commit or abort processing. If any CM detects what it considers a permanent communication or machine failure, it insures that all non-prepared transactions using the failed link or machine will eventually abort.

There are many subtleties to commit processing, including cycles in the participation graph, late requests that arrive at a server after it has become prepared, migration and/or replication of the coordinating site of a transaction, and optimizations that apply in certain cases. These issues are discussed elsewhere [10].

Log Manager. The QuickSilver *log manager* (LM) implements the abstraction of a record-oriented append-only file. TM uses the log to recoverably record the state transitions of transactions during the two-phase commit protocol. The file system uses the log to record changes to its metadata, in order to be able to atomically commit or abort a collection of changes. Long-running computations could use the log as a repository for checkpoints, completely independently of transactions.

LM accepts log records of arbitrary length from servers and buffers them in memory. A server may request that all records up to some point be *forced* to disk. The physical disk storage managed by LM is shared among all servers that use the log.

3 Advantages of Transactions

QuickSilver exploits transactions for a variety of purposes. These can be categorized as follows:

- guarding persistent data against inconsistencies in the event of failures,
- undoing a collection of changes,
- notifying servers of client termination, and
- synchronizing access to shared data.

Many of the advantages of using transactions in a general purpose system are described in an earlier paper [10]. In this section we explain each of these uses in more detail and illustrate them with examples of servers and application programs implemented in QuickSilver.

Failure atomicity. While data stored on a magnetic disk are not lost when a machine is rebooted or experiences a power loss, such data are still vulnerable to failures that occur while it is being modified. This is because partially completed updates may leave data in an inconsistent state. Database systems use atomic transactions to restore data consistency after a failure by undoing the effects of partial updates (i.e., transac-

tions in progress at the time of failure are aborted).

These techniques can also be very useful in a general purpose system. A text editor, for example, could lose part of a user's file if it crashed in the middle of writing a new version of the file to disk. Therefore, editors typically first save the old version of the file under a different name and delete it only after insuring that the new version has been written to disk (e.g. by calling `fsync` in Unix[3]). After a crash the user must start the editor with a special option to retrieve the old version of the file. Similar ad hoc recovery mechanisms are reimplemented within many applications. QuickSilver avoids this burden on applications by providing recovery in its file system. The QuickSilver *distributed file system* (DFS) supports transactional access to files and directories on local and remote machines [3]. DFS guarantees that updates of committed transactions are safely on disk and will not be lost due to subsequent failures. When a transaction aborts, DFS undoes all changes made to the file system by the transaction. Thus, all files that were modified by the transaction are returned to their original state, files that were created by the transaction are deleted, files that were deleted are restored, and files that were renamed or moved to a different directory are returned to their original directory or name.

Many QuickSilver applications rely on the atomicity guarantees provided by DFS. For example:

- Editors call `begin-transaction` and `commit` before and after writing a file to disk, respectively; there is no need to write a separate backup copy.
- The QuickSilver desktop utilities easily permit moving a subdirectory from one machine to another by clicking the mouse on a name in a directory window and dragging it to a different window. The benefits of this style of interface would be lost if users had to worry (even rarely) about cases when only part of the directory was moved or when some file might end up in two places or none at all. Since distributed commit guarantees atomicity across machines, this cannot happen in QuickSilver.
- Transactions are used to install a new version of the QuickSilver system files on a user's machine. This ensures that a crash during a system upgrade does not leave a machine in an inconsistent or unbootable state.
- QuickSilver provides a *parallel make* facility (`pmake`) that allows users to build several components of a large program in parallel using idle machines in our network. The `pmake` program reads a file containing a set of rules, each describing a command or sequence of commands for building one of the components of a program. A separate transaction is used for each such command sequence. This ensures that, when an instance of `pmake` is killed, no unwanted intermediate results (e.g. temporary files created by preprocessors such as `yacc` and `lex`) are left behind, while useful work (e.g. object files generated by a compile that completed before `pmake` was killed) is preserved. It also permits `pmake` to safely restart a sequence of commands elsewhere when one of the machines used by `pmake` crashes or is rebooted.

[3] Unix is a trademark of AT&T.

Undo. Transactional recovery mechanisms not only free programmers from worrying about failures, but also serve as an *undo* mechanism that simplifies application programming. For example, QuickSilver provides a *source control system* (`check`) for maintaining program source files on a file server. A user may `check-out` a set of files, thereby obtaining local working copies. After modifications have been made and tested, the files are transferred back to the file server (`check-in`). If any problem is detected during the `check-in` procedure, e.g. a file was not properly checked out, there is not enough space left on the file server, etc., `check` aborts its transaction. Any partial updates that may have been applied to source files or metadata on the file server or the user's machine are automatically undone.

Termination notification and cleanup. To support atomicity of updates across multiple machines, the QuickSilver transaction toolkit must implement a distributed commit protocol. By extending the notion of transaction to include servers that maintain only volatile state, it is possible to use transactions as a unified mechanism for notification and resource management throughout the system.

The following are some examples of volatile servers that use one-phase variants of the commit protocol to be notified when a program terminates:

- The window manager destroys all windows explicitly created by the program.
- The virtual terminal server closes terminal connections associated with the program (e.g. standard input and output).
- Taskmaster — the server responsible for creating processes and loading programs — destroys any child processes created by the program.

The advantage of using transactions for this purpose is that the same mechanism is used for all resources regardless of whether clients are local or remote. If ad hoc methods were used for each resource class, each such method would have to separately be extended to work in the distributed case. By using transactions for all resource recovery, the difficult issues of distribution can be implemented in TM once and for all.

The parent of a process started by Taskmaster, for instance, may be on a remote machine. A good example of this are commands started by pmake. If a user kills an instance of pmake while it is still in progress[4], all transactions created by pmake will be aborted. As a result, all commands running at remote machines at that time will be terminated as well, thus eliminating the problem of "orphans".

The parallel make facility provides another example of transactions as a distributed notification mechanism. To select machines for running remote computations, pmake interacts with the *remote execution scheduler*. This is a replicated server that runs on two or three designated machines and monitors the state of other machines on the same network. It accepts remote execution requests from pmake, waits until an idle machine is available, then tells pmake which machine to use to run a remote job. The scheduler bases its decision not only on such factors as CPU load and keyboard idle time, but also tracks how many remote jobs are currently running on behalf of a particular user, and where these jobs were started. The selection algorithm depends on the scheduler knowing when each remote job has terminated. For this purpose the scheduler participates as a one-phase server in the transaction pmake uses to run the remote job. This simplifies the scheduler interface (pmake does not need to make an explicit callback to the scheduler when a job ends), and ensures that the scheduler is notified even if the instance of pmake that started a remote job fails before the job ends.

Transactions as a unifying framework for all resource management simplify the problem of distributed notification, especially when more than two parties are involved, as in our example of parallel make (both pmake and the scheduler need to be notified when a remote job ends).

Concurrency control. Concurrency control techniques such as locking are used in database systems to synchronize access to shared data in order to avoid the problems of "lost updates", "dirty reads", and "unrepeatable reads" [8]. While most transaction systems use a locking policy that prevents lost updates, not all systems implement or enforce full serializability [1, 6]. In a general purpose system synchronization requirements vary greatly among applications. Therefore, QuickSilver allows each server to implement its own concurrency control policy.

[4]This is actually more frequent than one might think. If a header file contains an error, all modules that include that header file will fail to compile. Users typically kill a parallel make as soon as they notice such an error, to prevent a flood of essentially identical error messages.

Our choice of concurrency policy for the QuickSilver file system was driven by our desire to run standard Unix-like tools and applications. In such an environment, simultaneous use of a file system directory by unrelated programs is common; the /tmp directory on a Unix system provides an extreme example. It is also often desirable to be able to list the contents of a directory even if it has been modified by a transaction still in progress. For this reason DFS does not enforce full serializability. Instead, DFS obtains a lock on a directory only if the directory itself is renamed, created, or deleted. Write locks on individual directory entries prevent two transactions from renaming a file to the same name, for example, but read locks are not required to read a directory. Thus, it is possible for clients to read directory entries that have been changed by a transaction that may later abort. According to the terminology introduced by Gray et al. [6], DFS provides degree 1 consistency for directories.

The choice of locking policy for individual files is best illustrated by the example of a backup utility. If all files read by the backup program remained locked until the backup transaction ended, other programs that wanted to update those files would have to wait until the backup completed (potentially a very long time). On the other hand, it is undesirable to allow a file to be modified while it is being read by the backup program. Therefore, DFS uses both read and write locks to synchronize file access; read locks are released as soon as a file is closed, while write locks are held until transaction commit. This corresponds to degree 2 consistency for files (no lost updates and no dirty reads, but reads are not repeatable). Applications requiring degree 3 consistency (full serializability) for files can delay closing their files (and releasing the corresponding read locks) until they are ready to commit.

Because DFS holds locks on behalf of transactions, its locks can be shared among all processes (local or remote) participating in the transaction. In contrast, Unix associates file locks with a process on a particular machine.

Porting existing programs. We have given a number of examples of how transactions simplify writing new, potentially complicated distributed applications, especially when data consistency in the presence of failures is a concern. This leaves the question of how existing programs (e.g. programs ported from Unix) fit into a transactional system.

QuickSilver creates a *default transaction* for each process that it starts. This transaction commits when the program exits normally, or aborts if it terminates abnormally. The QuickSilver implementation of the C runtime library uses the TID of the default transaction

	Read-only		Read/Write		Unknown		Total	
Committed	151123	(89.37%)	11557	(6.83%)	0		162680	(96.21%)
Aborted	4507	(2.67%)	128	(0.08%)	0		4635	(2.74%)
Unknown	0		0		1775	(1.05%)	1775	(1.05%)
Total	155630	(92.04%)	11685	(6.91%)	1775	(1.05%)	169090	(100.0%)

Table 1: Number of transactions observed during a one week trace

in the IPC calls it generates. Hence, unless a program makes explicit calls to the transaction manager, all work done by the program is performed on behalf of its default transaction.

Sometimes it is desirable to run several programs under the same transaction, especially from a shell script. For this purpose we added transaction control commands to the shell. Users may explicitly create a new transaction that will be used as the default transaction for subsequent programs started from the shell. Other commands allow this transaction, and therefore the results of several programs, to be committed or aborted when desired.

As a measure of the success of default transactions, QuickSilver can run binary images of many common Unix utilities taken from AIX/RT[5], IBM's version of Unix for the RT-PC. These binary images obviously contain no code to deal with transactions, yet they behave atomically when run on QuickSilver.

4 Measurements of Transaction Usage

To better understand the actual usage patterns of transactions in the QuickSilver system, we traced all transactions that were created in our local network over a period of a week. To do this, we instrumented TM and DFS to generate timestamped trace records for every transaction. The information traced included

- the name of the program that created the transaction,
- the amount of file system activity performed on its behalf,
- a record of all transaction participants and their participation protocols, and
- the outcome (commit/abort) of the transaction.

[5] AIX is a trademark of IBM Corporation.

These trace records were buffered in memory, and periodically sent to a centralized trace server running on a dedicated machine. The trace server collected the trace buffers and saved them in a file. Data reduction was performed after tracing was completed.

During the week the trace was active, the QuickSilver user community used the system as normal, primarily for program development. The addition of transaction tracing introduced no noticeable overhead. While tracing was enabled, a total of 37 machines produced over 1.5 million trace records. Creations of 19419 tasks were observed, for at least 135 different programs, including everything from compilers to games to terminal emulation to telephone dialers. Table 1 summarizes the transactions traced and shows the distribution of committed vs. aborted and read-only vs. read/write transactions. DFS is currently the only two-phase recoverable server in QuickSilver, so transactions were considered read/write if they attempted to make any changes to a file system. Analysis of the trace data could not determine the status of some transactions (about 1%). These represent transactions still in progress when the trace ended, or buffered data lost when machines rebooted.

TM uses the "presumed abort" protocol [14] to control its distributed commit processing. For read-only transactions, this protocol requires no log records to be written, unlike the "presumed commit" or standard two-phase algorithms. Since most transactions used by QuickSilver are read-only (about 92%), this optimization is essential for good system performance.

Using the timestamps included in trace records we could also determine transaction lifetimes. The data showed that most transactions were not long-running. The median transaction lifetime was 0.2 seconds; 80% of all transactions ended within two seconds and 90% within 69 seconds.

Section 3 asserts that transactions are useful as a general purpose signalling mechanism for notifying servers of client termination. In support of this claim, Figure 2 shows the distribution of the number of servers noti-

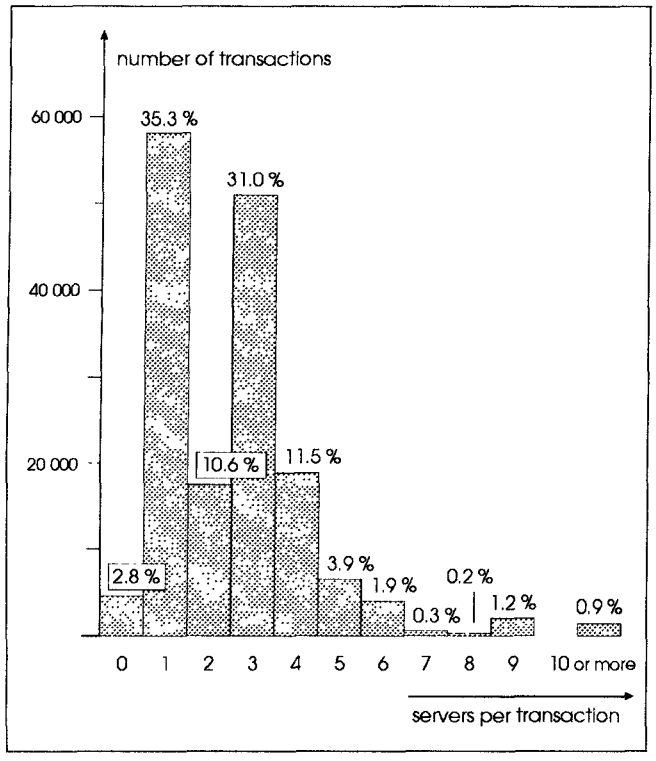

Figure 2: Number of servers per transaction

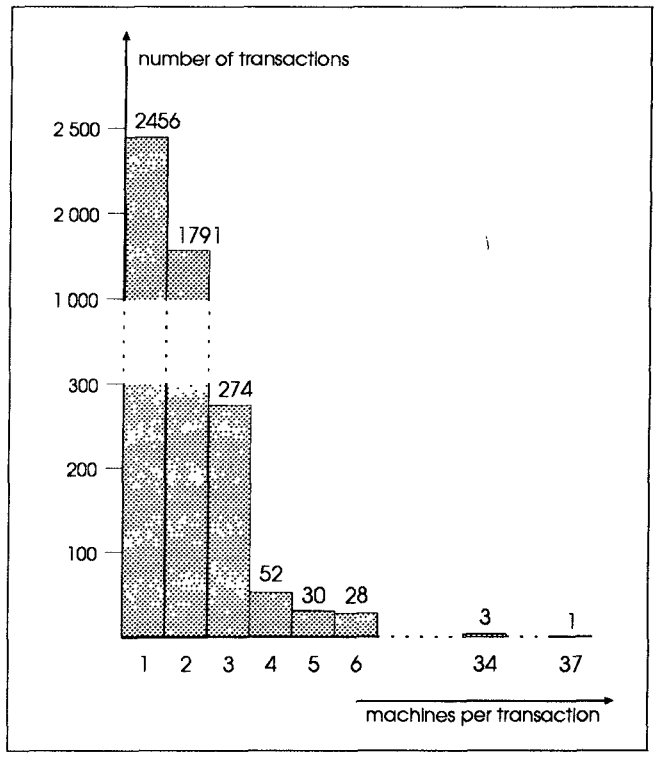

Figure 3: Number of machines per aborted transaction

fied during transaction termination processing. These counts do not include remote TMs, but do include the CMs on each machine that has transaction participants. The relative peak at three participants corresponds to transactions that had one remote participant, plus the participation of the two CM processes necessary for remote communication.

The first analysis of our trace data showed a high number of transactions (43132 or 21%) with no commit protocol participants, which therefore appear to perform no useful work. These transactions fall into two categories: unavoidable overhead inherent in the QuickSilver system, and transactions created unnecessarily due to inefficient or careless programming. The overwhelming majority of these transactions (89%) were of the latter type, caused by shortcomings in the shell and in pmake. Since these transactions obscured the results of our analysis, we filtered out trace records from transactions due to these two sources before calculating the figures presented in this section[6]. The remaining zero-participant transactions account for 2.7% of the total number of transactions (see Figure 2) and are due to the following legitimate sources:

- transactions used to contact only no-state servers, which do not participate in the commit protocol,
- transactions used to access only data that is replicated (as will be explained in Section 6, the standard I/O library transparently substitutes a separate top-level transaction when contacting replica servers), and
- transactions created by a program that is killed or otherwise terminates before interacting with any server using these transactions.

The average number of servers notified per transaction was 2.53. As many as 67 servers participated in a single transaction. There were 15 different servers that used the termination signalling feature of QuickSilver transactions. These range from servers mentioned already, such as the file system, window manager, and program loader, to more specialized applications providing print service, terminal emulation, desktop management, debugging services, and communication with foreign networks, among others. Together these facts indicate that the transaction mechanism is heavily used for server notification in QuickSilver.

The distribution of the number of machines involved in aborted transactions shown in Figure 3 illustrates the utility of transactions for failure cleanup. Clearly,

[6] A total of 38560 transactions, all of which were short-running committed transactions, were filtered out. Since these transactions had no participants, their only effect was to increase the total count of short, committed, single-machine, read-only transactions.

Test description	AIX time	QS time (1 txn)	QS time (sep txns)
Create 100 empty files locally	0.027 per file	0.065 per file	0.127 per file
Create 100 empty files remotely	0.054 per file	0.085 per file	0.187 per file
Create 32 256k byte files locally	1.160 per file	0.645 per file	1.012 per file
Create 32 256k byte files remotely	1.972 per file	1.752 per file	2.228 per file
Read 32 256k byte files locally	1.044 per file	0.666 per file	0.672 per file
Read 32 256k byte files remotely	1.338 per file	1.520 per file	1.554 per file
Run the null program 100 times	0.053 per run	–	0.098 per run

Table 2: Comparison of common operation under AIX and QuickSilver (times in seconds)

any mechanism for locating and freeing the resources held by a failed computation must be capable of scaling to fairly widely distributed computations. The most complicated transactions to abort were frequently commands started by `pmake`, although transactions of 65 different programs were observed to have aborted.

In the week we gathered transaction trace data, we observed the source code control system to abort `check-in` requests on 4 occasions, compared with 14 committed `check-in` operations. Both of these numbers are small, but they show that the undo capability of the QuickSilver file system is used in practice. The fact that undo is available permits the source code control application to be much simpler than it would have to be otherwise.

5 Effect of Transactions on System Performance

Our evaluation of the use of transactions in QuickSilver would not be complete without a discussion of how transaction support impacts system performance. Transactions were added to the QuickSilver operating system after many of its pieces were operational. Thus, we were able to observe IPC performance before and after introducing transaction participation monitoring. The difference measured just 5% for local IPC. Further, this difference is computed from a fairly small base. The round-trip time for a trivial local IPC request-response pair, including three system calls and two task switches, is about 350 microseconds on an RT-PC[7]. The round-trip time for a remote IPC is about 6.5 milliseconds on QuickSilver, which is not as good as other implementations on similar hardware [17]. The difference is not due to support for transactions, but rather to the high latency (1.3 ms per packet) of the RT-PC token ring adapter and to our implementation of CM as a process outside the kernel for software engineering reasons. We estimate the relative costs introduced into remote IPC by transaction support to be no more than those for local IPC.

The goal of this section, however, is not to provide a detailed analysis of the performance of various pieces of the QuickSilver transaction toolkit; such analysis has been presented elsewhere [10]. Rather the intent is to provide an idea of the overall effect of transactions on system performance.

QuickSilver is a complete system in that it has a community of users and hosts its own program development environment. In our use of the system, QuickSilver "feels" as responsive as a Unix system running similar applications on the same hardware, even though QuickSilver provides additional function, namely the atomicity guarantees of its file system. To attempt to quantify this subjective impression, we ran two performance tests on both QuickSilver and on Unix: we measured the time to perform representative simple operations and we ran the Andrew file system benchmark [11] to measure the performance of complex operations. The Unix tests were run against AIX version 2.2.1. Remote file operations under AIX used the NFS protocol. All machines used were RT-PC model 125s with 8Mb of memory. Disk controllers and network adapters were identical on all machines, and the same 4Mbit/sec token ring was used for all tests requiring more than one machine. The QuickSilver file system performance tests were run twice, once using a single transaction for the entire test, and once using individual transactions for each repetition in the test. The results of these tests appear in Tables 2 and 3.

The only file operation where AIX significantly outperforms QuickSilver is the local creation of small files. Due to the management of its buffer pool, AIX requires little or no disk I/O for this test. QuickSilver, however, forces file data to disk and file metadata to the log before committing transactions. These numbers give an indication

[7] An RT-PC model 125 is rated at between 4 and 5 Dhrystone MIPS.

	Local files		Remote files	
	AIX time	QS time	AIX time	QS time
MakeDir	3	1.5	5	2
Copy	14	16	33	25
ScanDir	30	27	35	42
ReadAll	41	56	57	91
Compile	328	334	376	348
Total time	416	434.5	506	517

Table 3: Andrew file system benchmark under AIX and QuickSilver (times in seconds)

of the pure transaction overhead, but have little practical impact on system performance, since few applications spend all their time creating small files. Also, files created and destroyed within a single transaction, such as temporary files created by the compiler, do not suffer this overhead. For operations on files of moderate size, QuickSilver is actually slightly faster than AIX in many cases. Our interpretation of these data is that QuickSilver is faster than AIX at moving data on and off disk, but slower at moving data across the network. Neither of these differences is related to support for transactions in QuickSilver.

A significant percentage overhead is also noticeable in the time to load and run a trivial program. Fortunately, the absolute magnitude of the time difference is small. The time to load real programs is dominated by the time spent reading the programs from disk or across the network.

There are several other points of interest in Table 2. The time to read files is barely affected by whether QuickSilver uses one transaction for the entire test or one transaction per file. Since these transactions are read-only, no log activity is required, and DFS can use the *commit-read-only* optimization of the presumed abort protocol [14], which will exclude DFS from the second phase of the protocol, since it does not need to know whether the transaction commits or aborts. In contrast, using one transaction per file written has a large effect on the running time of the tests. There are a number of reasons for this: (1) Log forces must occur for each file, rather than a single log force at the conclusion of the test. (2) The log is on one of the disks used for storing files, so additional seeks are required. (3) The forcing of file data blocks occurs synchronously rather than asynchronously as the file system buffer pool fills.

The Andrew benchmark (Table 3) creates a directory tree (*MakeDir*), copies a set of files into the directory tree (*Copy*), runs `ls` to list the status of all files (*ScanDir*), scans each file by running `grep` and `wc` (*ReadAll*), and compiles and links all files (*Make*). The last phase of the benchmark is CPU bound; since AIX has a different linker than QuickSilver, we only ran the compile part of the *Make* phase using identical binaries for the compiler on AIX and QuickSilver. The overall results of the benchmark show the QuickSilver performance to be within 5% of AIX in the local and 2% in the remote case. Only for the ReadAll phase of the benchmark does AIX significantly outperform QuickSilver (27% faster locally, 37% remotely). Since the benchmark runs a separate instance of `grep` and `wc` for each individual file[8], the difference between QuickSilver and AIX in the local case can be attributed to faster program loading under AIX. In the remote case, AIX has an added advantage due to NFS client caching. We chose not to implement client caching of remote files in DFS because cache consistency protocols were not a focus of the project.

Many factors other than the presence or absence of transactional behavior affect the times reported in Tables 2 and 3:

- Buffer pool sizes, buffer replacement strategies, and disk layout algorithms differ between the two file systems.
- AIX has a more sophisticated virtual memory manager than QuickSilver, allowing it to reuse the pages of a previously loaded program without refetching them from the file system.
- All tests were run on a shared token ring during normal working hours.
- The file systems used for the test had been in use for months or years; therefore, their free space was fragmented.

Despite all these caveats, we believe that our measurements give an indication of the overall impact of transactions on system performance and support our claim that making programs behave atomically does not cause a significant increase in the cost of common operations.

[8] In fact, the benchmark spends more than half of its time in this phase repeatedly loading `grep` and `wc`.

6 Lessons Learned

Over the past several years, we have accumulated a great deal of experience with building and using applications and servers that rely on transactions. Most of that experience has been positive; transactions have proven to be a very useful abstraction for writing robust distributed systems. In those cases where we encountered problems, they were due to limitations in our implementation of transactions, not due to the transaction concept itself. In this section we present some of the lessons we learned from building and using the QuickSilver system.

Lesson 1: *Writing transactional applications is simple.*

Most QuickSilver client programs see little or no complexity added by transactions. The use of default transactions together with facilities for manipulating them from shell scripts permits even programs that contain no transactional code to behave atomically. Most short-running application programs fall into this category.

Small changes must be made to long-running and interactive programs in order for them to exhibit correct transactional behavior. For such programs, it is necessary to identify recoverable units of work and perform them within shorter transactions. For example, consider retrieving files using the file transfer utility `ftp`. If all files transferred by `ftp` were created under the default transaction of the `ftp` daemon, the locking policies of DFS would prevent the files from being accessed until the `ftp` daemon exitted. Worse yet, all of these files would automatically be deleted if the `ftp` daemon crashed or its machine was rebooted. Similarly, a text editor needs to create a new transaction each time it writes a file back to disk. To make this as simple as possible, QuickSilver provides a library to push and pop the current default transaction. These routines add a means of controlling transactions to a standard I/O interface, in our case the C `stdio` library. An alternative would be to change the interfaces of all library functions to include an explicit transaction parameter. This would work better for programs using multiple threads, each of which might need its own transaction. Our approach, however, is better suited to porting existing applications, almost all of which are single-threaded anyway.

In most cases it has been straightforward to identify the recoverable units of work within applications that need to be performed as transactions. Some of the more non-traditional uses of transactions as a notification mechanism, however, require more careful design. The following example illustrates this. In earlier versions of the system, termination of the default transaction associated with a process was used to signal Taskmaster to destroy that process. This caused problems when we added the facilities to the shell that allow more than one process to run under the same default transaction; processes associated with a sequence of commands started from a shell script were not destroyed until the whole command sequence committed. The reason for this problem was that the default transaction was used for too many different purposes. The obvious solution was to associate another transaction with each process to be used exclusively for notifying servers that must take actions when a process terminates.

Lesson 2: *Writing simple transactional servers is simple; writing complex transactional servers is difficult but worthwhile.*

The effort involved in making a server follow the QuickSilver transaction protocols varies greatly, depending on how much benefit of the transaction toolkit the server intends to make available to its clients. At one extreme is a server like the window manager. It simply records the TID associated with the IPC request that created each window, and destroys the window when it receives a commit or abort notification for that TID from TM. This is comparable to the work required of an X server to close windows when TCP sockets it uses for interactions with its clients are signalled that the sending process has disappeared.

At the other extreme is DFS, which uses all of the services of the transaction toolkit to provide its clients with the ability to undo changes to individual files. A substantial part of this server is devoted to providing transactional atomicity [2]. The code complexity of the file system is due directly to the design choice we made to provide atomicity. We elected to build such a file system because we felt it provided a useful level of function, but a non-atomic file system is also a possibility for QuickSilver. Indeed, the precursor of the present file system used the transaction mechanism only for signalling that open files should be closed. The implementation overhead due to transactions in this older file system was similar to that in the window manager.

Generally, we have found it easier to write robust applications for QuickSilver than the corresponding applications for Unix. The reason for this is that the code necessary to handle failures in a distributed system has been moved out of applications and into servers, making even "quick-and-dirty" applications with no recovery code at all likely to behave reasonably in the face of failures. Since there are more application programs than servers, the use of transactions is a net win.

Lesson 3: *We survived without nested transactions, but they would have been useful in some cases.*

Since QuickSilver does not provide support for *nested transactions* [15], the failure of a single participant in a transaction causes the whole transaction to abort. While this facilitates automatic, comprehensive cleanup in the presence of failures, it can make it more difficult for applications to recover from partial failures and continue operating.

The parallel make utility illustrates this. When a command started by `pmake` on a remote machine fails because the machine reboots, it makes sense to retry the command on a different machine, rather than aborting `pmake`. For this reason `pmake` uses a separate transaction for each set of commands it starts on a remote machine. Each such transaction must be committed as soon as its remote commands have finished, so that DFS will release write locks on files that need to be accessible as input to subsequent steps of the computation. A consequence of using separate transactions is that when the `pmake` program is killed, only output from computation steps that were in progress at that time is cleaned up; output from earlier completed steps is preserved. This behavior is desirable for the typical uses of `pmake` for program development. But `pmake` also has other uses; it can, for example, be used to apply an upgrade to a software package installed on a user's machine. In this case it is clearly desirable to undo all of the effects of `pmake` if it does not complete successfully. Our current version of parallel make cannot be used for such applications. Nested transactions would solve this problem; running each step of a computation as a nested subtransaction of `pmake` would allow `pmake` to recover from partial failures while committing all its updates atomically.

There are several other examples in QuickSilver where nested transactions would be more useful than separate top-level transactions:

- In Section 3 we pointed out that transactional undo can simplify applications even if failures are not a concern (e.g. `check`). Transactions as an undo mechanism would be more generally useful if an application could undo only part of its updates. Separate transactions cannot be used for this if the application needs to commit all its updates as one transaction. Nested transactions would provide a more powerful undo facility.
- The QuickSilver standard I/O library supports transparent access to replicated files and directories in read-only mode. Because a client transaction needs to be able to commit even if one of the replica servers fails, separate transactions must be created to access remote replicas. This method can be used to read but not to update replicated data atomically. Using nested transactions instead of separate top-level transactions would solve at least part of this problem. Updates to replicated data could be committed as one transaction, while allowing a transaction to succeed even if one of the replica servers fails. However, extra effort would still be needed to bring a server that has missed some updates up-to-date after it recovers.

Lesson 4: *Long running update transactions are not (or should not be) a problem.*

Long running transactions that update files are a potential source of difficulty, since files can remain locked for a long time. We found that in practice such transactions are not a problem, because of the following observations:

1. Updates performed by long-running or interactive programs can in most cases be encapsulated in shorter transactions using the techniques described under Lesson 1.
2. In cases where updates cannot be broken into shorter transactions, the data written by a long-running transaction generally do not need to be accessible to other applications before the transaction ends.

The canonical example of the second class of applications is a backup utility. Backing up a large directory tree may take a long time, but the fact that the archive file created by the backup cannot be read by other transactions until the backup is completed does not impact other programs.

While the long duration of an update transaction is not a problem, the fact that long transactions may make arbitrarily large updates to file systems has uncovered a major shortcoming in the log manager. Such large update transactions require arbitrarily large amounts of space in the log. The current implementation of LM supports only a single on-line log partition on disk, which it manages as a circular buffer with no provisions for garbage collection or archiving to off-line storage. The fact that only a finite amount of space is available for the log artificially restricts the size of transactions that make modifications to the file system. This has distorted the use of transactions in QuickSilver in some cases, forcing us to modify utilities to use multiple small transactions rather than a single large one, even though the utility should behave atomically. The program that installs a new version of the operating system is an example of such a utility.

One of the surprises in building and using the QuickSilver system has been the complexity required in an industrial strength log subsystem. A major effort is now

underway to build a new log service capable of garbage collecting its on-line storage and archiving either to network log servers or to off-line storage [9].

Lesson 5: *Long running read-only transactions need not be a problem.*

As we observed above, transactions that run for a long time and update a large number of files are rare. Long-running transactions that *read* many files, on the other hand, are very common. Read locks held by such transactions are a potential source of problems, since a read lock prevents other transactions from updating a file. Because of our desire to run many standard Unix tools and applications without modification, we decided to solve this problem by giving up strict serializability as the default concurrency policy of the QuickSilver file system. As explained in Section 3, DFS releases the read lock on a file as soon as the client closes the file. We found this policy sufficient to avoid problems associated with read locks, because almost all long running applications close a file soon after reading it. The only exception we encountered was an interactive utility for browsing files too large to view with a text editor. In this case we added code to the browser that buffers large pieces of a file in memory and closes and reopens the file between reads, in order to avoid frequent lock conflicts.

In our current implementation, however, a long-running read-only transaction may cause problems even if it does not keep files open. The window manager provides an example of this phenomenon. Once started, it runs until its machine is switched off or rebooted. Hence its default transaction remains active for weeks or months at a time. The window manager reads data, such as bitmaps for display fonts, from a remote file server. Each file is closed immediately after reading to avoid interfering with utilities that update font files. Nevertheless, the file server remains a participant in the window manager's transaction. This has two consequences:

1. The session between CM on the window manager's machine and CM on the file server is kept open, causing additional background message traffic.
2. State associated with the transaction continues to occupy space in non-pageable kernel tables on the file server machine.

Besides the window manager, there are several other long-running programs that run on all machines and read remote files. As a result, kernel tables on the file server machine can grow and eventually overflow, causing the server to crash. To avoid this problem we modified library routines used by the shell, the window manager, the user interface toolkit, etc., to create separate short transactions when searching for and reading files. While fixing the immediate problem, this solution is unsatisfactory for two reasons. First of all, the additional transactions that are created often do not correspond to any meaningful units of work in the application (this is in contrast to our earlier mentioned changes for avoiding long-running update transactions). A symptom of this is that it was much more difficult to track down all the places in application code that needed to be modified because files were read from file servers. Secondly, to minimize the changes to application code, most of our modifications are embedded in library routines. As a result, more transactions than necessary are created. For example, as many as eight read-only transactions may be created and committed to start up a window application from a shell. This adds unnecessary overhead to starting programs under QuickSilver.

A better solution would be to enhance the transaction management and communication protocols as follows. Using timeouts, CM could detect transactions that have been inactive for extended periods of time. The transaction manager would then attempt to remove the remote machine from its list of participants. This is only possible for read-only transactions, so TM would need to call the remote transaction manager, which in turn would ask each local participating server if the transaction needed to be kept active. If all servers on a machine agreed, the transaction could be removed from the kernel tables at the remote machine. If no more active transactions required the CM communication session between two machines, CM would close the session. In this way, long-running programs could simply use their default transaction for reading files from the server. The kernel, TM, CM, and DFS would cooperate to periodically reclaim inactive resources without requiring new transactions. This would eliminate the need for modifications to library routines and application code.

Lesson 6: *A flexible concurrency control policy allowing a wide range of consistency options is desirable.*

The QuickSilver philosophy regarding concurrency control is similar to the one adopted for recovery. Different concurrency control policies and serialization algorithms are appropriate for different kinds of servers. For recovery QuickSilver provides a log service that is useful for a variety of recovery algorithms and leaves the choice of a particular algorithm to the implementer of the service. Concurrency control is more difficult; the choice of the best concurrency control policy depends to a much greater extent not only on the kind of services provided by a server, but also on how these services will be used by its clients (e.g. short transactions vs. long-lived ones).

In the case of DFS our goal was to provide a file system with an interface that would allow us to run stan-

dard Unix tools and applications. As we described in Section 3, we made specific choices for the default concurrency control policy of DFS that we felt would be appropriate for most applications. Some applications need more synchronization, while in other cases an even less restrictive locking policy would have been more convenient. An example of the latter case is the problem of debugging a new program. If a software interrupt (e.g. illegal memory address) occurs during a test run of the program, QuickSilver starts up an interactive debugger without killing the program. This allows the user to inspect the state of the program in order to identify the problem. However, since transactions created by the program are still active at this point, output files written by the program remain locked and cannot be read by the user. This applies in particular to trace files containing information written by the program as a debugging aid! For this case it would be desirable for DFS to allow read access to a file without obtaining a lock. An alternative solution might be for the debugger to inherit the transactions created by the program. However, any tool used to examine the output of the program (e.g. the `grep` utility or an editor) would have to be started from within the debugger to become a participant in the necessary transactions. Furthermore, the program might have created several transactions, making it difficult to determine which transaction to use to examine a particular output file.

Since DFS releases read locks when a file is closed, an application that reads a set of files is not guaranteed to be serializable with respect to update transactions. An application that needs to see a consistent snapshot of the file system, for example a utility for installing a new version of a software package from a file server onto a user's machine, must keep all files it reads open until it commits. Alternatively, an application can effectively lock a whole subtree of a file system by temporarily renaming the directory at the root of the subtree. A cleaner, more general solution, however, would be to extend the DFS interface to allow clients to specify, for example, that read locks are to be held after a file is closed until the transaction ends, or — to solve the debugging problem — that no read locks are to be obtained.

Another problem related to concurrency control is the possibility of deadlocks. In QuickSilver, we avoid deadlocks by never blocking client requests in case of lock conflicts; DFS completes a request with an error code instead of waiting for a lock held by another transaction to be released. A disadvantage of our approach is that sometimes requests are completed with an error code even when no deadlock is present. An alternative would be to block client requests on lock conflicts and use a deadlock detector [16] to abort transactions when necessary, as do some database systems. We rejected this alternative for two reasons. First of all, an interactive program waiting for user input may prevent other transactions from making progress even if there is no cycle in the wait-for graph. Secondly, there is no good way of deciding which transaction to abort when a deadlock is detected. Therefore, we believe it is better to let clients decide what actions to take in case of a lock conflict. Our approach could be improved by letting servers block a request for a limited time while waiting for locks to be released. By waiting until a timeout expired before completing a request with an error return code, the server would reduce the probability that clients observe lock conflicts.

7 Related Work

The concept of transactions has been accepted and used in database systems for many years. Both local [7] and distributed [12] database systems have been implemented that provide the failure atomicity, recoverability, and isolation properties of transactions. These concepts have also been extended to specific servers provided by operating systems. Locus [18] provides a transactional file system, for example.

More recently, several projects have offered transactions as a more general operating system mechanism for recovery services. Avalon [5] and Argus [13], for example, are languages that hide the details of implementing recoverable or distributed programs within new high-level programming language constructs. The Camelot system [5] provides similar functions for the C language through macros and library routines.

QuickSilver differs from the above systems in several ways. Lightweight extensions to the basic two-phase commit protocol allow QuickSilver to use transactions as a basis for all recovery management in the system, including servers that manage volatile state. Through its use of default transactions for programs and its support for transactions at the lowest level in the system, namely IPC, QuickSilver is able to make all programs in the system benefit from the use of transactions. Transactions are available to programs written in conventional programming languages, and, with little or no modification, to programs ported from systems that do not provide transactions. We have found the latter properties invaluable in building a system complete enough to support its own development.

Servers in QuickSilver must provide their own recovery code, which makes writing them more difficult than in a language that handles recovery. On the other hand, by making the facilities needed to implement recovery

available as a toolkit, QuickSilver servers may take advantage of the semantics of their operations to optimize recovery algorithms. DFS, for example, supports concurrency at the level of individual bits in its recoverable space allocation component.

8 Conclusions

The QuickSilver distributed system uses transactions pervasively:

- All interprocess communication is done on behalf of transactions.
- All programs run under one or more transactions.
- All updates to persistent data in the file system behave atomically with respect to failures.

In database systems transactions are used to provide atomicity of updates, isolation between multiple updaters, and permanence of committed updates. Our experience with QuickSilver has shown that all of these properties are useful in a general-purpose system as well; they aid synchronizing access to shared data and help maintain consistency of persistent, distributed data in the presence of failures. Transactions also provide an undo mechanism to applications that simplifies programming even when failures are not a concern. Beyond these traditional uses of transactions, lightweight variants of the two-phase commit protocol allow transactions to be used successfully throughout the system as a unifying mechanism for distributed notification and resource management.

We found that writing transactional applications is not difficult. In fact, transaction support in the operating system makes it easier to write distributed applications (e.g. `pmake`) and applications that must maintain data consistency (e.g. `check`). Also, porting existing applications to a transactional environment is not difficult; many Unix tools run unchanged under QuickSilver. Because of the transactional file system and the use of default transactions, such programs behave atomically under QuickSilver. Writing transactional servers can be more complex, but the additional effort is offset by the advantage of simpler client code.

Applications that use transactions only for purposes provided by different means in other operating systems (e.g. termination notification) suffer no or only minimal additional cost due to the presence of transactions. Tracking transaction participants, for example, adds only about 5% to the cost of local IPC. Where transactions provide additional functions, in particular automatic cleanup and recovery by the file system, they impose only a small performance overhead. In these cases the benefits of simpler, more robust applications outweigh the performance cost. Overall, we found the performance of the QuickSilver system to be comparable to a non-transactional system running on the same hardware.

Our experience with building and using transactional programs over several years has been mostly positive. Although there are a few areas in which our implementation needs to be improved, we found no inherent problems with the transaction model itself. In particular, we found that long-running transactions do not pose a problem in a general purpose system, provided that (1) flexible concurrency control policies allow clients to use just the amount of concurrency control they need, and that (2) the log service supports transactions that write large amounts of log data.

In summary, we have shown that the transaction mechanism as implemented in the QuickSilver operating system provides a particularly powerful means for solving many of the problems introduced by operating system extensibility and distribution. This power has a reasonable cost; practical experience with a complete system based on transactions shows both the implementation complexity and performance penalty of transactions to be small.

9 Acknowledgements

Many people have contributed to the QuickSilver project. We will not attempt a complete list, but the following people have at one time or another all worked on the project for at least several years: Luis-Felipe Cabrera, Mike Goodfellow, Roger Haskin, Yoni Malachi, Dan McNabb, Jon Reinke, Wayne Sawdon, and Marvin Theimer.

References

[1] M. M Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[2] Luis-Felipe Cabrera, John McPherson, Peter Schwarz, and Jim Wyllie. A comparison of two log-based implementations of atomicity. In preparation, 1991.

[3] Luis-Felipe Cabrera and Jim Wyllie. QuickSilver distributed file services: an architecture for horizontal growth. In *Proceedings of the 2nd IEEE conference on computer workstations, Santa Clara, CA*, March 1988.

[4] David R. Cheriton. The V kernel: a software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[5] Jeffrey L. Eppinger, Lilly Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kauffmann, 1991.

[6] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems*, pages 365–394. North Holland Publishing Company, 1976.

[7] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[8] J.N. Gray. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmueller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1979.

[9] Roger Haskin, Dean Daniels, Wayne Sawdon, Daniel McNabb, and Jon Reinke. The QuickSilver recovery log service. In preparation, 1991.

[10] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[11] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[12] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems*, 2(1):24–38, February 1984.

[13] Barbara Liskov et. al. Argus reference manual. Technical Report MIT/LCS/TR-400, MIT, November 1987.

[14] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of process model of distributed transactions. *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 76–88, August 1983.

[15] Elliot B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MIT Press, 1985.

[16] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.

[17] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[18] Matthew J. Weinstein, Thomas W. Page, Jr., Brian K. Livezey, and Gerald J. Popek. Transactions and synchronization in a distributed operating system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 115–126, December 1985.