Palo Alto Research Center

The Alpine File System

Mark R. Brown, Karen Kolling, and Edward A. Taft

XEROX

The Alpine File System

Mark R. Brown, Karen Kolling, and Edward A. Taft

CSL-84-4 October 1984 [P84-00046]

Abstract: Alpine is a file system that supports atomic transactions and is designed to operate as a service on a computer internet. Alpine's primary purpose is to store files that represent databases; an important secondary goal is to store ordinary files representing documents, program modules, and the like.

Unlike other file servers described in the literature, Alpine uses a log-based technique to implement atomic file update. Another unusual aspect of Alpine is that it performs all communication via a general-purpose remote procedure call facility. Both of these decisions have worked out well. This paper describes Alpine's design and implementation, and evaluates the system in light of our experience to date.

Alpine is written in Cedar, a strongly typed modular programming language that includes garbage-collected storage. This paper reports on using the Cedar language and programming environment to develop Alpine.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

The system we describe here is known within Xerox as "Research Alpine" or "RAlpine". To aid readability we shall call it "Alpine" in this paper.

Alpine is a file system that supports atomic transactions and is designed to operate as a service on a computer internet. (Svobodova's survey article [Svobodova, 1984] describes the concept of atomic transactions and the notion of file service on a computer internet.) Alpine's primary purpose is to store files that represent databases; an important secondary goal is to store ordinary files representing documents, program modules, and the like.

Today (in February, 1984) an Alpine server stores the personal and shared databases used daily by about 45 people at Xerox PARC; it has been available as a service for ten months. The system does not yet meet all of its goals, and is still being developed.

This paper is organized as follows. Section 2 describes why we built Alpine. As one might expect, our rationale for building a new file server had a large impact on its design goals. Section 3 details Alpine's aspiration level along many dimensions: concurrency, reliability, availability, and so forth. We try to justify these goals in terms of the motivations from Section 2.

Section 4 discusses the two design decisions that had the greatest impact on the Alpine system: the use of logs instead of shadow-pages to implement atomic file update, and the use of remote procedure calls instead of streams or message passing for communication. It may seem unnatural to discuss an implementation technique, like logging, before the design of Alpine has been described in more detail. But both of the decisions highlighted in Section 4 were made very early in the design of Alpine, and have had a large impact on the rest of the design. In Section 5 we present the design in detail, explaining the basic abstractions that a programmer sees when using Alpine. Section 6 discusses our implementation of these abstractions.

Section 7 evaluates Alpine. We describe the experience gained from Alpine applications, give the current status and plans for the system, and present the results of running some simple benchmarks.

Alpine is written in the Cedar language [Lampson, 1984], using the Cedar programming environment [Teitelman, 1984]. Section 8 of the paper reports our experiences in using Cedar to develop Alpine.

At this point in the paper, only a few facts about Cedar are relevant. The *Cedar language* evolved from Mesa [Mitchell *et al*, 1979], a programming language featuring interface and implementation modules, strong type checking, and inexpensive concurrent processes. (Ada is similar to Mesa in many respects.) The Cedar language's most significant extensions of Mesa are garbage collection and the option of runtime, rather than compile-time or bind-time, type discrimination and checking. The *Cedar nucleus* is a basic environment for running Cedar programs. It includes a paged virtual memory, the runtime support for incremental garbage collection, and a simple file system with a read-write interface. There is no protection boundary between other Cedar programs and the Cedar nucleus; the Cedar language's type checking makes it very unlikely that a non-malicious program will corrupt the nucleus.

2. Motivation for Building Alpine

Among the research topics being investigated in Xerox PARC's Computer Science Laboratory (CSL) are database management systems and their application to information management. We feel that in the future, databases will supplement or replace hierarchical directory systems as the primary tool for organizing computer files and that more and more information will migrate away from files with specialized formats into well-integrated databases. In our environment, all shared files, including files that represent databases, are stored on file servers, so research involving shared databases requires an appropriate file server. Early in 1981, when the Alpine project began, we had two file servers available to us, IFS and XDFS [Israel *et al*, 1978, Mitchell and Dion, 1981].

IFS (Interim File Server) stores nearly all of our laboratory's shared files. IFS was developed in 1977 and is designed for transferring entire files to and from a personal workstation's local file system. It does not support random access to files; this makes it inappropriate for storing databases. IFS is written in BCPL for the Alto and would not be easy to modify.

XDFS (Xerox Distributed File System, also known as Juniper) is a file server with more ambitious goals. XDFS supports random access to files and provides atomic transactions. We constructed a simple database management system [Brown *et al*, 1981] and several applications that used XDFS to store shared databases. We found that using XDFS crippled some important applications. The basic operations of creating a transaction, reading and writing file pages, and committing a transaction were disappointingly slow. The server crashed frequently, and recovery from any crash took over an hour. And the server allowed only a small number of randomly distributed file pages to be updated in a single transaction (although a much larger number of sequentially clustered pages could be updated). At that time the Cedar programming environment project was well under way and XDFS, written in Mesa for the Alto, was an unattractive basis for further development.

Since neither IFS nor XDFS met the need, we decided to build a new Cedar-based file server, Alpine, to support our database research. This server would subsume the functions of XDFS, which could be decommissioned.

Given that we were building a new file server, we considered making it good for storing ordinary files as well as databases. One reason for this is that Cedar runs on a personal workstation that is several times faster than the Alto, and the time spent waiting for IFS to do its work (or retrying when the server was rejecting connections) was increasingly noticeable. A Cedar-based Alpine server could use the same hardware as the fastest Cedar workstation. Also, Cedar planned to include a local file system that supported transparent caching of remote files. We expected that the more convenient access to remote files would increase the load on our IFS servers, making the performance mismatch between workstation and server even greater. We also knew that extensions to IFS in support of transparent file caching would be desirable.

For all of these reasons, we hoped that Alpine could replace IFS as well XDFS. This was definitely a secondary goal, because the deficiencies of IFS were not as serious as those of XDFS.

For instance, by partitioning our laboratory's files between two IFS servers we significantly improved file server response to Cedar workstations.

We would not have considered building Alpine on an Alto. IFS seems to represent the limit of what we can build on a computer with 16-bit addressing for data and no hardware support for virtual memory. Many of the problems with XDFS can be traced back to Alto limitations. Cedar's large virtual and real memory are what allowed Alpine's goals to be a more ambitious than those of IFS.

3. Design Goals

We require the following terminology for the discussion that follows, and in the remainder of the paper. A *user* is a human being who uses a computer service, for example, by making mouse clicks or typing commands to a command interpreter. A *client* is a program that uses a computer service, i.e., by making local or remote procedure calls.

The purpose of Alpine was to support other CSL research projects; Alpine was not an end in itself. This had some noticeable effects. It encouraged a conservative design – we wished to use proven techniques from the published literature, where possible, and to use existing code, where possible. (IFS, our model of a successful file server, was built using many existing Alto packages.) It also encouraged a decomposition of the problem in a way that allows the system to begin supporting its primary clients, database systems, without requiring that the entire file system be complete. We have a lot to learn about system support for databases, and our clients have a lot to learn about using shared, remote databases; the sooner we could begin to provide database storage service, the sooner both parties could get on with the learning. At the same time, Alpine was itself a research project that permitted us to try out a set of ideas on what functions should be included in a file server, and how these functions are best implemented.

Any project to build a file server must arrive at decisions about a wide variety of issues, including concurrency, reliability, availability, file access, distribution of function, capacity, access control, and accounting. We set the following design goals for Alpine:

Concurrency and reliability. We decided that Alpine should implement atomic transactions on files. Alpine's aspiration level for concurrency should be to allow concurrent update of one file by several transactions. Alpine's aspiration level for reliability should be to support configurations that survive any single failure of the disk storage medium without loss of committed information, but this degree of redundancy should not be required in all Alpine configurations.

Providing atomic transactions at the level of file actions like "read a page" and "write a page" is unusual. A typical database system implements a set of access methods as a layer on top of a conventional file system. (Examples of access methods include database records accessible by sequential or hashed searching, B-Tree indices on records, and direct pointer chains linking records.) Most database systems implement transactions at the access method level. We can identify three potential advantages for implementing transactions at the file level.

First, solving concurrency control and recovery problems in a file system simplifies any database system that uses it. Without file-level transactions, each access method must implement transactions by explicitly setting locks, writing recovery information on the disk, and interpreting this information in case of both soft and hard failures. The basic atomic action available to the access method implementor is writing a single page to disk.

Simplifying the database system is especially important if the database system itself is an object of research. Database systems that were originally designed for business applications are now being tried in new domains such as office systems and computer-aided design. This experience has suggested ways to make database systems more useful in these new domains, but the complexity of real database systems has hindered experimentation with them.

Second, file-level transactions provide the option of running the database system on a separate machine from the file system. This moves processing load away from the machine that is shared by all users of a particular database. Most database management systems are processor-bound, so it is worth considering the option of moving some processing load from the file server to the workstation, or from the file server to a closely coupled database server.

Running the database system on a separate machine from the file server may also have some drawbacks. It does not minimize communication because it transmits file pages rather than database records or user keystrokes and display updates. This would certainly be a problem on a low-speed communications network. If the database system runs in the workstation, it is unlikely to be secure, and hence it can only provide ironclad access control at the level of files, not database record sets or fields. In CSL, neither of these factors rules out locating the file system and database system on different machines.

Third, providing transactions at the file level ensures that an application can use the same transaction abstraction to deal consistently with all data stored in the file system. A reasonable application may well wish to manipulate data stored both in a raw file and in some database used to index this file, or to manipulate data stored in several specialized database systems. It is possible to create several database systems that share the same transaction abstraction, but it seems more likely to occur if the transaction implementation is shared, as it can be with file-level transactions.

Accompanying the three potential advantages of file-level transactions is one potential disadvantage—a loss of concurrency due to locking of units such as files or pages that are unrelated to the semantics of the database. A database system that performs its own concurrency control and recovery can set locks on logical units, and avoid some artificial conflicts that arise from file or page locking. Our intuition is that the loss of concurrency would be unacceptable in, for example, an online banking application, but would not be noticeable in many of the applications that we find most interesting, such as office systems, programming environments, and computer-aided design. If this proves false, then we have no choice but to move the responsibility for concurrency control into the database system. Even in this case not much is lost as long as most of the implementation of file-level transactions carries over to the database system.

Availability. Since storage medium failure is rare, we decided that Alpine should be allowed to recover from it slowly (in a few hours); crashes caused by software bugs or transient hardware problems may be more frequent, and recovery should therefore be much faster (5-10 minutes). It should be possible to move a disk volume from one file system to another and to store a volume offline. Our environment did not demand continuous availability of a file server; we could tolerate scheduled downtime during off hours and small amounts of downtime due to crashes during working hours. If better availability had significant cost, we didn't want it.

File access. Alpine should be efficient at both random and sequential access to files. Access at the granularity of pages (fixed-length aligned byte sequences) seemed sufficient for databases; it is no great advantage to provide access to arbitrary byte sequences. The goal is good average-case performance for file access, not the guaranteed real time response that, for instance, a voice file server must provide.

Distribution of function. We decided that moving an Alpine volume from one server to another should be transparent to clients, and that a single transaction should be able to include files from several different Alpine servers. A volume-location service seemed easy to implement using the Grapevine registration database, whereas a file-location service based on Grapevine would be too slow. Multi-machine transactions are now supported by several commercial database systems, and adequate implementation techniques are well understood.

Capacity. We thought it would be nice if one Alpine server could maintain hundreds of simultaneous workstation connections; the cost of an inactive connection should be low at the server end.

Access control. We decided that Alpine should implement simple controls over access to files. Our aspirations for access control were low because we felt that database systems will implement their own access control policies; in this situation, it is the database system, not the user of the database system, that requires authenticated access to files.

Accounting. We decided that Alpine should implement simple controls over the allocation of disk space to various users and projects. This is what IFS provided, and it was acceptable. Using the file server to store databases does not create new problems in this area.

Because our Cedar workstations are powerful personal computers with their own rigid disks, the issue arose of whether Alpine should be able to run on a Cedar personal workstation. This supports the storage of local databases, and small system configurations that do not contain a dedicated file server. We decided that this is useful and Alpine should allow it. Alpine will not serve as the only file system on the workstation; the standard file system is still required for performing special functions such as storing boot files. As a result, allowing this configuration adds almost nothing to Alpine's implementation. A workstation-based Alpine file system may be limited in some ways when compared with a server, for instance, by the lack of independent disk volumes to improve performance, reliability, and availability.

We excluded several things from Alpine, deciding that they were best regarded as clients of Alpine rather than part of Alpine itself:

A directory system (mapping text names for files into internal file system names) is required in order to make Alpine usable, but a directory system supporting databases need not aspire to replace IFS. At some future time, we may wish to build a distributed directory system to support location-independent file access and perhaps some form of file replication. So there are advantages in avoiding a strong coupling between Alpine and its directory system.

A Pup-FTP server [Boggs *et al*, 1980] is a client of the directory system and of Alpine. The FTP server is not required for database access, but is necessary to replace IFS.

A system for archiving files from the primary disk storage of a file server and for bringing back archived files is a client of the directory system and of Alpine. IFS does not have an archiving system, either.

4. Two Major Design Decisions

4.1 Implement Atomic File Update Using a Log

The two techniques most often discussed for implementing atomic file update are logs and shadow-pages. A paper on the System R recovery manager [Gray *et al*, 1981] argues that large shared databases are best updated using the log technique, and many database systems use it. We chose the log technique for the Alpine file server. But the published literature shows that all transaction-based file servers have chosen the shadow-page technique [Svobodova, 1984]. What are the tradeoffs between shadow-pages and logs?

To describe the shadow-page technique, we must clarify some terminology concerning files. To us, a file *page* is simply a fixed-length aligned byte sequence in a file; if the page size is 512 bytes, then bytes [0 .. 511] are one page, and bytes [512 .. 1023] are the next. A disk *sector* is a region of a disk that is capable of storing a page value. A file system implements files using disk sectors.

The shadow-page technique is based on the observation that the sectors of a disk are always referenced through a level of indirection contained within the file system. To read a file page, the client presents a file ID and page number, and the file system reads some disk sector and returns its contents. The file system controls the mapping from the pair [file ID, page number] to disk sector; call this the *file map*. The file map must be represented on the disk in some recoverable way, i.e., it must survive file system crashes.

Using this observation, one technique that the file system may use to update a page is to allocate a new sector, write the new page value there, and modify the file map so that reads go to the new sector rather than to the original. If the client commits the transaction containing the page update, this change to the file map must be made permanent on the disk, and the sector containing the previous value must be freed. If the client aborts the transaction, then no permanent change to the file map needs to be made, and the new sector must be freed.

There are many variations of the shadow-page idea. They largely concern the management of two permanent data structures: the file map and the *allocation map*, which represents the free or occupied status of each disk page. Various representations of the file and allocation maps are possible, and for each representation there are many different algorithms that may be used for updating the maps in a way that is consistent with the outcome of transactions.

The log technique, on the other hand, represents each file as a file in a simple file system that does not support transactions. A request to update a file page is represented by a record in a log, generally common to all files in a single file system. In one variation, called the *redo log*, a log record contains the file ID and page number of the page being updated, plus the new value of the page. The new value of the page is not written to the underlying file before transaction commit; instead, a volatile version of the file map is updated to say that the current value of the page resides in a specific log record. To commit a transaction, one writes a commit log record for the transaction, then waits until all log records for the transaction have reached the disk. (Something analogous to this commit record is also required in the shadow-page technique, but we did not point this out above.) After a transaction commits, a background process copies the new page value from the log to the underlying file and erases the volatile file map update. A request to abort a transaction is implemented by erasing the volatile file map update. There are many variations of the log idea in addition to the redo log; Lampson's paper [Lampson, 1983] contains a more general view of logs and their uses.

The log itself is represented as a fixed-length file, and this file is managed as a queue of variable-length records. The storage for the oldest log record may be reused when there is no volatile file map entry pointing to it, that is, when its transaction either has aborted or has committed and written the contents of the log record to the file updated by the transaction.

To illustrate the differences between the two techniques, we consider a client that creates a transaction, writes ten pages to an existing file under this transaction, and finally commits the transaction. Compare the I/O activity required in a shadow-page implementation with that in a log implementation:

> Before commit, the shadow-page implementation allocates ten disk sectors and writes new page values to them. It also writes enough additional information to disk so that the correct changes can be made in the file and allocation maps regardless of whether the transaction commits or aborts. After the transaction commits the new sectors are incorporated into the existing non-volatile file map and the ten existing sectors are freed.
>
> Before commit, the log implementation writes ten log records, each slightly larger than a file page, to the log, and forces the log to disk. After commit, it reads data from the log and writes it to the underlying file. (In fact the file writes are buffered, and are only forced to disk periodically, unlike log writes that are forced for each transaction.)

Which technique is preferable? The shadow-page implementation seems to require less I/O, because each page is written just once, whereas the log implementation writes data first to the log,

then later reads data from the log and writes to the underlying file. It is true that the shadow-page implementation must also update the non-volatile file map, which is not necessary in the log implementation, but this should require fewer than ten page writes because many data pages are referenced by each file map page.

Now consider the following six advantages of the log implementation. They consist of four reasons why the log implementation might perform better, one added function it provides, and one implementation advantage.

The first performance advantage is that it allows an extent-based file structure, that is, a file structure that attempts to allocate logically sequential pages in a physically sequential manner on the disk. An extent-based file structure has performance advantages for both sequential and random access. The advantage for sequential access is obvious: the extent-based file starts contiguous and stays that way, while the shadow-page file loses its contiguity when it is updated. The advantage for random access derives from the fact that the file map is smaller in an extent-based file structure than in a shadow-page file structure. The file map is smaller because it only needs to represent the location of a few long runs of pages, rather than representing the location of each page individually. With a smaller file map, it is less often the case than a file read causes extra I/Os to read the file map.

The second performance advantage is that many of the I/Os in the log implementation are cheap. The log is implemented as a single file and log writes are strictly sequential. Existing database systems show how to optimize log writes [Peterson and Strickland, 1983]. Many clients will commit a transaction shortly after performing all of the transaction's writes. In this case, the log records containing the new values of the updated pages will still reside in the buffer pool after the transaction commits. This means that reading these log records, which is necessary in order to write the data to the underlying file, uses no I/Os. So under these assumptions, each updated page is written once to the log (a sequential I/O) and once to the underlying file (a sequential or random I/O depending upon the client's access pattern).

The third performance advantage is that the log implementation performs fewer disk allocations. Log records are allocated sequentially from a bounded file; this involves a simple pointer increment and test. Shadow pages, in contrast, are allocated using the general disk allocator for file pages—after all, a shadow-page becomes part of a file when the transaction commits. So the allocation of a shadow-page is more complex and expensive than the allocation of a log record. The requirement that disk storage allocated to shadow-pages not be lost in a system crash increases the complexity and expense.

The fourth performance advantage is that the log implementation (at least the redo log we have described) defers more work until after commit. This improves average response time for updates if the system is lightly loaded, or heavily loaded with a bursty pattern of update requests. If the load is steady and heavy, there is still a performance advantage if part of the update load is concentrated on a small set of pages. With the redo log implementation, a heavily updated page is written to its underlying file only when necessary to reclaim log space; otherwise its current value resides in the buffer pool and in the log.

The added function that the log implementation supports is file backup. Backup protects against failure of the disk storage medium, e.g., due to a head crash, by keeping a redundant copy of all committed data. The whole point of the shadow-page scheme is to write the new data just once, but this is not consistent with a desire to maintain a backup copy. In a log-based file implementation, backup can be implemented by taking a periodic dump of the file system and saving all log records written after the most recent dump. It is even possible to take a dump concurrently with file update activity, so that backup is not disruptive.

The major implementation advantage of the log-based approach to atomic file update is that it can be layered on an existing simple file implementation. This is true because the log approach does not rely on any special properties of how files are implemented; it simply reads and writes files. The faster the underlying file system, the better the log-based implementation performs. The log-based transaction implementation can be structured so that most of it does not depend on details of the underlying file system, and is easy to move from one file system to another. Adding transaction logging to a good existing file system is much less work than writing a new file system from scratch, including all of its associated utility programs.

4.2 Communicate Using Remote Procedure Calls

Many specialized protocols have been designed for access to a file on a remote server. For instance, XDFS included a specially tailored protocol built directly upon the Pup packet level [Boggs *et al.* 1980]. In principle, clients of XDFS viewed interactions with the file server in terms of the sequence of request and result messages exchanged.

In practice, clients of XDFS used a standard package to provide a procedure-oriented interface to the file server. One of the usual arguments in favor of message-oriented communication is that messages provide parallelism: a client program can send a message and then do more computation, perhaps even send more messages, before waiting for a reply. In the Mesa environment parallelism can be obtained by forking processes, and these processes are quite inexpensive [Lampson and Redell, 1980]. So in the Mesa environment it was natural to view interactions with XDFS as remote procedure calls.

Unfortunately, the implementation of XDFS itself did not reflect this view (perhaps because the Mesa process facilities were added after the project was well under way). In particular, there was no clean separation between the file system and data communications components of XDFS. For instance, a data type defining the packet format was known to the file system component of XDFS. As a result, changes in the communications component could force recompilation of the entire system.

In Alpine we enforce a clean separation between the file system and data communications components. This is done by defining the system's behavior in terms of a small set of Cedar interfaces, *not* in terms of any specific network protocol. The interfaces are usable either by a client that needs all the features of Alpine file service or by one that implements a simplified file access protocol like Pup-FTP.

We were quite fortunate that a project to produce a general-purpose remote procedure call facility [Birrell and Nelson, 1983] was carried out concurrently with the design and implementation of Alpine. One result of this project is a compiler, called *Lupine*, that takes a Cedar interface as input and produces server and client *stub modules* as output. The client stub module exports the Cedar interface, translating local calls into a suitable sequence of packets addressed to a server; the server stub imports the Cedar interface, receiving packets and translating them into local calls. (We discuss this more fully in Section 5.1.) Lupine imposes some restrictions on interface design; it will not transmit arbitrary data structures from one machine to another. Even so, Lupine did not inhibit the design of our file system interfaces. Lupine has made the evolution of Alpine much simpler, because we can build a new version of the system without investing a lot of effort to keep the communications component consistent with the file system component. Lupine generates about 6000 lines of code for use with Alpine. It is possible that we could have maintained the clean separation between communications and file system without Lupine, and even without adopting an RPC-style interface, but the existence of Lupine reduced the possibility of getting this wrong.

Is a general-purpose RPC less efficient than a protocol that is specialized to the task at hand? Not necessarily. Birrell and Nelson take advantage of the assumption that RPC is the standard mode of communication by optimizing the path of a remote call and return through the layers of communication software. It would not be feasible to optimize more than a few protocols in this way, so it follows that the processor overhead for an RPC can be less than for the more specialized protocol. After normalizing for the effects of using different processors, Cedar RPC is about four times faster than XDFS in making a simple remote procedure call. With some specialized microcode or hardware, the general-purpose RPC can still be speeded up by a factor of three or more.

5. Design

An Alpine file system exports four public interfaces: AlpineTransaction, AlpineVolume, AlpineFile, and AlpineOwner. The most effective way of describing Alpine's design is in terms of these interfaces.

In most respects the descriptions below match the Cedar interfaces actually exported by Alpine. We have chosen to omit some features of the actual interfaces that are unimplemented, or are implemented but have not been exercised by any client.

Our notation for procedure declarations is

ProcName [arg1, arg2, ..., argN] −> [result1, result2, ..., resultM]
  ! exception1, exception2, exceptionK.

Normally an argument or result consists of a simple name; when the type for the argument or result is not obvious, the name is followed by ":" and the type. An exception is a name optionally followed by "{", then a list of the possible error codes for the exception, and then "}".

5.1 Cedar RPC Background

The Grapevine system [Birrell *et al*, 1982] provides a name space for users of the Pup internet. A Grapevine name for an individual, such as "Schroeder.pa", or for a group, such as "CSL↑.pa", is called an *RName*. Cedar, and in particular the Cedar RPC machinery, uses RNames to name individuals and groups.

A Cedar *interface* is essentially a record type whose named components are procedures. An *interface instance* is a record value containing named procedure values. So to obtain the value of procedure *P* in interface instance *I* one writes *I.P*, and to apply this procedure to parameters *args* one writes *I.P[args]*. Normally, a Cedar program module gets an interface instance by being linked together with another program module that produces the value. The Cedar RPC system allows a client program on one machine to obtain an interface instance on another machine, often a server. To make this happen, the server first calls the RPC system to declare that its interface instance is available to remote clients. This gives the interface instance a name (an RName), which is generally just the server's name. For example, an instance of an Alpine interface is named by an RName in the "alpine" registry, such as "Luther.alpine". A client program then calls the RPC system, specifying both the interface (type of service) and instance name (server) desired. The client program can make calls through this interface instance using the same syntax as for local calls. So in this style of communication the identity of the server is implicit in the interface instance, and is not specified as a parameter to each procedure.

A client must call the RPC system to establish a *conversation* with a server before it can make authenticated calls to the server. The conversation embodies the authenticated identities of both the client and the server, and the security level (encrypted or unencrypted) of their communication. By convention, a conversation is the first parameter to all server procedures. In our descriptions of Alpine public procedures below we shall omit the conversation parameter.

5.2 AlpineTransaction Interface

The AlpineTransaction interface contains procedures for the creation and manipulation of transactions. All Alpine procedures that manipulate data require a transaction.

There are two distinct services behind the AlpineTransaction interface: the transaction coordinator and the transaction worker. The coordinator manufactures transaction IDs and plays the role of master in two-phase commit. The worker performs data manipulation under a transaction and plays the role of slave in two-phase commit. Ideally this level of detail would be hidden from clients, but to do this requires that a worker be able to communicate with a coordinator of a transaction, given only the transaction ID. We were not willing to implement this as part of Alpine, so clients of multi-server transactions must do more work; see CreateWorker below.

A transaction ID contains enough unpredictable bits so that it is extremely difficult to forge, and can be treated as a capability. Any client that knows the transaction ID may participate in the transaction.

It is useful for a server administrator to be able to bypass access control checks in some situations, while obeying them in normal situations. We allow an administrator to explicitly "enable" a specific transaction for access control bypassing.

Create [createLocalWorker: BOOLEAN] -> [transID]
! OperationFailed {busy};
Call to coordinator; requests the coordinator to create a new transaction. If *createLocalWorker*, the coordinator calls *CreateWorker[transID*, RName-of-this-server*]*. Raises *OperationFailed[busy]* if the server is overloaded.

CreateWorker [transID, coordinator: RName]
! Unknown {coordinator, transID};
Call to worker; informs the worker that transaction *transID* is being coordinated by the Alpine instance *coordinator*. Raises *Unknown[coordinator]* if the worker cannot communicate with *coordinator*, and *Unknown[transID]* if *transID* is unknown to *coordinator*. If this procedure returns without an exception, the worker is willing to perform Alpine data manipulation procedures using this transaction.

Finish [transID, requestedOutcome: {abort, commit}, continue: BOOLEAN] -> [outcome: {abort, commit, unknown}, newTransID];
Call to coordinator; requests the coordinator to complete the transaction. When a client using a transaction has completed its calls to the Alpine data manipulation procedures, it calls with *requestedOutcome = commit* to commit the transaction. To give up on a transaction and undo its effects, a client calls with *requestedOutcome = abort*. If the transaction has already committed or aborted, this procedure simply returns the outcome, which is *unknown* if the transaction has been finished for more than a few minutes.
A call with *continue = TRUE* commits the effects of a transaction, then creates and returns a new transaction that holds the same data state (locks, open files, and so forth) as the previous transaction. System R calls this a *save point* [Gray *et al*, 1981]. This feature encourages a client that is making many updates to commit periodically, whenever it has made a consistent set of changes.

AssertAlpineWheel [transID, enable: BOOLEAN]
! OperationFailed {grapevineNotResponding, notAlpineWheel}, Unknown {transID};
Call to worker. When called with *enable = TRUE*, causes subsequent Alpine procedure calls for this (conversation, *transID*) pair to pass access control checks without actually performing the checks; raises *OperationFailed[notAlpineWheel]* if the caller is not a member of the AlpineWheels group for this server. When called with *enable* = FALSE, causes normal access control checking to resume for this (conversation, transID) pair. Raises *Unknown[transID]* if *transID* is not known to the worker. This might be because the client has not called CreateWorker, or because the transaction is already finished.

5.3 AlpineVolume Interface

A *volume* is an abstraction of a physical disk volume. There are several reasons for having a volume abstraction. If a volume corresponds to a disk arm, then clients have the ability to improve performance by distributing the most-frequently accessed files across several disk arms. A volume may also be the unit of scavenging, so limiting the size of volumes sets and bounds the time to restart if a single volume needs to be scavenged. Alpine does not have any ambitions of its own for volumes; it just uses whatever volume structure is provided by the underlying simple file system. Alpine does assume that volume IDs are globally unique.

A server often contains several volumes that are equivalent from the point of view of its users. In particular, a user may not care on which volume he creates a file. For this reason, an Alpine file system's volumes are partitioned into one or more *volume groups.* A group contains one or more volumes, and every volume belongs to exactly one group. Disk space quotas for owners are maintained for volume groups rather than individual volumes; see the AlpineOwner interface, below. When creating a file a client may specify a volume group, rather than a specific volume. Alpine then selects some volume in the group and creates the file there; see AlpineFile.Create below.

Any procedure that takes an *transID* parameter raises the exception *Unknown[transID]* if the *transID* is not known to the worker. This might be because the client has not called AlpineTransaction.CreateWorker, or because the transaction is already finished. We shall not mention this exception in the individual procedure descriptions for any of the procedures in this or the next two sections.

GetNextGroup [transID, previousVolumeGroupID] -> [volumeGroupID]

! Unknown {volumeGroupID};

Stateless enumerator for the on-line volume groups of this Alpine instance. Calling with *previousGroup = nullVolumeGroupID* starts an enumeration, and *volumeGroupID = nullVolumeGroupID* is returned at the end of an enumeration.

GetGroup [transID, volumeGroupID] -> [volumes: LIST OF VolumeID]

! Unknown {volumeGroupID};

Returns the list of volumes belonging to the specified volume group.

5.4 AlpineFile Interface

A *file* is a set of property values and a sequence of 512 byte pages, indexed from zero. Each file is contained in a single volume. A file is named by a file ID that is unique within the containing volume. A UniversalFile is a globally unique file name: a [volume ID, file ID] pair.

The set of file properties is fixed (not expandable). It is described by the enumerated type *Property*, with elements *byteLength*, *createdTime*, *textName*, *owner*, *readAccessList*, *modifyAccessList*, and *highWaterMark*.

Alpine does not interpret a file's byte length, created time, or text name. These properties are read and written by clients and Alpine performs no checking.

A file *owner* is an entity identified by the RName of a person or a group. A database indexed by the set of valid owner names, called the *owner database*, is associated with each volume group. The disk space occupied by a file is charged against its owner's space quota in the volume group containing the file.

Access to a file is controlled by its *read access list* and its *modify access list*. Each of these is a list of RNames and the two special names "Owner" and "World". For implementation simplicity the total number of characters in these lists has a fixed upper bound, though the distribution of characters between the lists is not controlled. This restriction on list length is less severe than it might seem because any list may contain groups with an unbounded number of members. An Alpine user may read a file if he is either in the read or the modify list and may modify a file if he is in the modify list. An owner database entry contains a similar access list, the *owner create list*, that is used when creating new files for an owner. The usage of the owner create list is described later along with the Create procedure.

A file may contain pages at the end whose contents are considered undefined by Alpine. When a file is created all of its pages have undefined contents, and when a file is extended the additional pages have undefined contents. The boundary between the defined and undefined portions of a file is called its *high water mark*. Precisely, the high water mark is the page number of the first page with undefined contents. The high water mark allows Alpine to optimize updates to the undefined portion of the file. The high water mark is automatically maintained by Alpine as pages of a file are written, but a client can also set the high water mark explicitly. Decreasing the high water mark is a way of declaring that some portion (or all) of a file's contents are no longer interesting.

To manipulate a file a client must first *open* it. Opening a file accomplishes three things. First, it sets a lock on the file (locks are discussed below). Second, it associates a brief handle, the *open-file ID*, with a considerable amount of state that is kept by the server and need not be supplied in each call. That is, an open-file ID represents a single client's access to a single file under a single transaction. Third, it performs file access control, and associates a level of access (read-only or read-write) with the open file. This allows a client to restrict itself to read-only operations on a file even if the client has permission to open the file for writing.

The procedures that transfer page values to and from open files operate on runs of consecutive pages, allowing clients to deal conveniently in pages that are any multiple of the basic page size, and reducing the per page overhead on sequential transfers. A run of consecutive pages is described by a PageRun: a [PageNumber, PageCount] pair.

Alpine uses conventional locks [Gray, 1978] to control concurrent access to files from different transactions. Alpine implements both whole-file and page locking. When page locking is used on a file, the file properties are locked as a unit. Instead of hiding these facts from clients, we allow clients to influence the locking.

The choice between whole-file locking (the default) and page locking is made by the client at the time a file is opened. Locking is also performed implicitly as a side effect of read and write operations. A read access locks the object in read mode (the default) or write mode, while a modify access locks the object in write mode. A client who is satisfied with the default behavior need not be concerned further with locks. But more ambitious clients may specify page granularity locking, may explicitly set locks in order to reserve resources, and may explicitly release read locks before the end of a transaction to reduce conflicts.

A number of the Alpine procedures take a *lockOption* parameter, which is used to specify the desired lock mode and to indicate what Alpine should do in case of a conflict in trying to set the lock. If there is a conflict and *lockOption.ifConflict* is *fail*, the procedure raises the exception *LockFailed*. If *lockOption.ifConflict* is *wait*, the procedure does not return until either the lock is set or the transaction is aborted.

Alpine places few restrictions upon its clients when it comes to concurrent operations within a single transaction. Locks do not synchronize concurrent procedure calls for the same transaction; this is the client's responsibility. It is perfectly OK to have, for example, seven *ReadPages* and two *WritePages* calls in progress at the same time for one transaction. It is probably not useful for a read to be concurrent with a write of the same page; the result of the read will be either the old value or the new value of the page, but the reader cannot tell which. A request to commit a transaction while another Alpine operation is in progress for the transaction is a client error and aborts the transaction.

Any procedure that takes an *owner* or *volumeID* parameter raises the exception *Unknown* {*owner*, *volumeID*} if the corresponding argument does not identify an existing object. Any procedure raises the exception *StaticallyInvalid* if it detects an out of range argument value by some static test, for instance, an invalid value from an enumerated type. Any procedure with a *lockOption* parameter raises the exception *LockFailed* if *lockOption.ifConflict* is *fail* and the lock cannot be set. We shall not mention these exceptions in the individual procedure descriptions in this or the next section.

Create [transID, volumeID, owner: RName, initialPageCount, referencePattern: {random, sequential}] −> [openFileID, universalFile]

! AccessFailed {ownerCreate, spaceQuota}, OperationFailed {insufficientSpace}.

Creates and opens the new file *universalFile* with *initialPageCount* pages under *transID*. *volumeID* may be either a specific volume ID or a volume group ID; in the latter case, the server chooses among the volumes of the group. The client must be a member of *owner*'s create access list, and the allocation must not exceed *owner*'s quota or the capacity of the server.

All file properties are set to default values: *byteLength* and *highWaterMark*: 0, *createdTime*: now, *readAccess*: "World", *modifyAccess*: "Owner" plus *owner*'s create access list, owner: *owner*, and *textName*: empty string.

If the call is successful, the new file is locked in *write* mode, and the client has read-write

access to the file using *openFileID*. Alpine uses the *referencePattern* parameter to control read-ahead and buffer replacement strategies for the file.

Open [transID, universalFile, access: {readOnly, readWrite}, lock, referencePattern: {random, sequential}] -> [openFileID, fileID]
! AccessFailed {fileRead, fileModify}.
Opens an existing file *universalFile* for access under *transID*. The file's access lists are checked to ensure that the client is allowed to use the file as specified by *access*. If *access* = *readOnly*, the client is restricted to read operations on *openFileID*, even if the client passes the checks for read-write access. The entire file is locked in *lock.mode*; *lock.ifConflict* is also remembered and is used when performing certain file actions (e.g., Delete) that do not allow a LockMode specification. Alpine uses the *referencePattern* parameter to control read-ahead and buffer replacement strategies for the file.
A file ID consists of both the file identifier, which is permanently unique, and some location information, which is treated as a hint [Lampson, 1983] and is subject to change during the lifetime of the file. *Open* returns a file ID whose location hint may differ from the hint in the file ID that is contained in *universalFile*. When the file ID changes, the client should store it in its own data or directory structures so that subsequent *Open* calls will be more efficient.

Close [openFileID]
Breaks the association between *openFileID* and its file. The number of simultaneous open files permitted on one Alpine file system is large but not unlimited; clients are encouraged to close files that are no longer needed, particularly when referencing many files under one transaction. Closing a file does not terminate the transaction and does not release any locks on the file; nor does it restrict the client's ability to later re-open the file under the same transaction.

Delete [openFileID]
! AccessFailed {handleReadWrite};
First locks the entire file in write mode; then deletes the file. *openFileID* must allow read-write access. *Delete* makes *openFileID* invalid for subsequent operations. The owner's allocation is credited with the released pages when the transaction commits.

ReadPages [openFileID, pageRun, resultPageBuffer, lockOption]
! OperationFailed {nonexistentFilePage}.
Reads data from the pages described by *pageRun* of the file associated with *openFileID*, and puts it contiguously into client memory in the block described by *resultPageBuffer*.
A *ResultPageBuffer* consists of a pointer into virtual memory and a word count; Lupine understands that the contents of the block (not the pointer and count) is the real value, and that this value is really a procedure result (not an argument).

WritePages [openFileID, pageRun, valuePageBuffer, lockOption]

! AccessFailed {handleReadWrite}, OperationFailed {nonexistentFilePage}.

Writes data from client memory in the block described by *valuePageBuffer* to the pages described by *pageRun* of the file associated with *openFileID*, which must allow read-write access.

A ValuePageBuffer has the same representation as a ResultPageBuffer, but Lupine understands that in this case the value is really a procedure argument (not a result).

LockPages [openFileID, pageRun, lockOption]

Explicitly sets locks on the specified pages of the file.

UnlockPages [openFileID, pageRun]

If the specified pages of the file are locked in *read* mode, then removes those locks. It is the client's responsibility to assure consistency of any subsequent operations whose behavior depends on the data that was read under those locks. Lock requests are counted and the lock is not removed until one UnlockPages has been done for each LockPages or ReadPages previously performed on the same pages. Attempts to remove nonexistent locks or locks other than *read* are ignored without error indication.

GetSize [openFileID, lockOption] –> [size: PageCount]

Returns the file's size, after setting a lock on the size property.

SetSize [openFileID, newPageCount, lockOption]

! AccessFailed {handleReadWrite, spaceQuota}, OperationFailed {insufficientSpace}.

Sets a write lock on the file properties, then changes the file's size to *newPageCount*. Decreasing a file's size locks the entire file in write mode. *openFileID* must allow read-write access; the client need not be a member of the owner's create access list.

If the file size is being increased, the owner's allocation is charged for the new pages immediately, but if the size is being decreased, the owner's allocation is credited when the transaction commits.

ReadProperties [openFileID, desiredProperties: PropertySet, lockOption] –> [properties: LIST OF PropertyValuePair]

The type *PropertySet* represents a set of file properties, for example {*createdTime*, *textName*}. The type *PropertyValuePair* represents a property, paired with a legal value for that property, for example *[createdTime, February 12, 1984 11:46:52 am PST]* or *[textName, "/Luther.alpine/MBrown.pa/Walnut.segment"]*. *ReadProperties* returns a list of the property values specified by *desiredProperties*, ordered as in the declaration of *Property*.

WriteProperties [openFileID, properties: LIST OF PropertyValuePair, lockOption]

! AccessFailed {handleReadWrite, ownerCreate, spaceQuota}, OperationFailed {insufficientSpace, unwritableProperty};

Writes the supplied properties, which may be ordered arbitrarily.

To write the *byteLength*, *createdTime*, *highWaterMark*, and *textName* properties requires that the *openFileID* have *access = readWrite*. To write *readAccess* or *modifyAccess* requires that the client be the file's owner or a member of the owner create list for the file's owner, but does *not* require that *openFileID* have *access = readWrite*. To write *owner* requires that both the above conditions be satisfied, and additionally requires that the client be a member of the owner create list for the new owner; the disk space occupied by the file is credited to the old owner and charged to the new one.

If there is insufficient space in the leader page to represent the new properties, the exception *OperationFailed[insufficientSpace]* is raised. If multiple properties are written by one call and an error occurs, some of the properties may nevertheless have been written successfully.

5.5 AlpineOwner Interface

The *owner database*, a database indexed by the set of valid owner names, is associated with each volume group. The set of entries in an owner database record is fixed, and is described by the enumerated type OwnerProperty, with elements *spaceInUse*, *quota*, *rootFile*, *createAccessList*, and *modifyAccessList*.

An owner's *spaceInUse* is the total number of pages in all owned files, plus a fixed overhead charge per owned file. Hence, spaceInUse is changed as a side effect of *AlpineFile.Create*, *Delete*, *SetSize*, and *WriteProperties* (writing the *owner*) calls. An owner's *quota* is an upper bound on spaceInUse, enforced by *AlpineFile.Create* and *SetSize* calls. Creation of files for an owner is governed by the *owner create access list*, as described above.

The owner *root file* is a *UniversalFile* that can be read and written by clients. It is intended for use by a directory system.

The creation of files for an owner is governed by the *owner create access list*, as described in the previous section. Modification of the owner create access list is controlled by the *owner modify access list*. This allows for a limited amount of decentralized administration of the owner database.

Alpine allows other updates to the owner database, such as creating owners and changing space quotas, only for transactions with wheel status enabled. We omit the procedures that perform these administrative updates.

ReadProperties [transID, volumeGroupID, owner: RName, desiredProperties:
OwnerPropertySet] -> [properties: LIST OF OwnerPropertyValuePair]

Reads the properties specified by *desiredProperties*, ordered as in the declaration of *OwnerProperty*.

WriteProperties [transID, volumeGroupID, owner: RName, properties: LIST OF
OwnerPropertyValuePair]
! AccessFailed {alpineWheel, ownerEntry}, OperationFailed {ownerRecordFull,
grapevineNotResponding};

Writes the supplied properties, leaving the others unchanged.

The *spaceInUse* property is read only. The owner, and members of the owner's modify list, can update the *createAccessList* of an owner record. The owner and members of the owner's create list can update the *rootFile* property of an owner record. If an update is restricted to these properties and the access control checks fail, then WriteProperties raises the exception *AccessFailed[ownerEntry]*. Otherwise the update requires the client to be enabled as a wheel using *transID*.

Alpine enforces some integrity constraints: owner access lists always contain "Owner" and never contain "World". An update to an owner access list may raise *OperationFailed [ownerRecordFull]*, much as for file access lists.

6. Implementation

Alpine's implementation decomposes into six components, which we shall call Buffer, Log, Lock, Trans, File, and OwnerDB.

Buffer provides a buffer manager abstraction [Gray, 1978] of the underlying simple file system. Operations in its interface allow runs of consecutive pages to be specified, but Buffer does not always return the entire run in contiguous virtual memory. Buffer always reads the disk in blocks several pages long, but writes only the dirty pages back. Buffer allows its clients to present hints to control file read-ahead and write-behind. Because we do not update file pages until after commit, there is no need for Buffer to provide fine synchronization between writing dirty file pages to the disk and writing log records to the disk. For the implementation of log checkpoints, Buffer provides the operation of synchronously writing all currently dirty pages, but this is allowed to take a relatively long time and does not interfere with other Buffer operations that may dirty more pages.

Log provides a simple log abstraction. It is built upon Buffer in a straightforward way. It requires the ability to write a specified page run to disk synchronously.

Lock implements a conventional lock manager [Gray, 1978]. Lock detects deadlocks within a single file system and uses timeouts to break the deadlocks that it cannot detect.

Trans implements a conventional transaction manager [Lindsay *et al*, 1979]. Inter-server communication for coordinating transactions is performed with RPC. The implementation avoids redundant synchronous log writes in the common case of a transaction that performs no file updates, and in the case of a transaction that performs all file updates in the same Alpine instance as the transaction coordinator.

File implements the file abstraction. The implementation of atomic file update generally follows the redo log scheme discussed in Section 4.1. Each file contains a leader page, used to represent file properties. This requires length restrictions on variable-length file properties, but allows a straightforward implementation.

Alpine deviates from the pure redo log scheme in the following important respect: actions that allocate disk space, such as create-file or extend-file, are not deferred until after transaction commit. Instead, Alpine writes a log record that is sufficient to undo the action, then performs it. Should the transaction abort, the undo is performed. One reason that we do not defer disk allocations is that with the underlying file systems we expect to use, there is no practical way to commit to a disk allocation without actually doing it. And unlike a file page write, a disk allocation requires very little information to be written into the log in order to make it undoable.

Another deviation from the pure redo log scheme is the file high water mark optimization. Page writes past the high water mark are made directly to the file and not logged, and the high water mark is advanced. All file updates must be forced to disk before a transaction commits. If a transaction aborts the high water mark must not be advanced.

OwnerDB implements the owner database and access controls. The owner database is represented as a normal Alpine file; to read and write the owner database, OwnerDB calls File through the *AlpineFile* interface. For instance, when a client calls *AlpineFile.Create* to create a ten page file for a particular owner under a transaction, *Create* calls OwnerDB to read the owner database under this transaction and verify that the client is authorized to create the file and that the ten additional pages will not exceed the limit for the owner. To reduce contention OwnerDB builds a volatile representation of the quota and releases the read lock on the owner database file page. Just before the transaction commits, OwnerDB updates the owner database under the transaction to show ten more pages in use by the owner. Hence, the transaction mechanism guarantees the consistency of the owner database. The owner database file is organized as an open-address hash table with one record per page; when it fills up, the administrator calls a procedure to expand and reorganize the owner database, again using the transaction facility to cover the update.

OwnerDB tests membership in access control lists by calling Grapevine. The efficiency of this is made acceptable by caching the results of these calls. In order to track Grapevine updates it suffices to discard old cache entries, since Grapevine's own consistency guarantees are not strong.

7. Experience and Evaluation

By early 1983, we had coded and started to test an Alpine implementation that lacked two-copy logging and incremental backup (both are needed to meet the goal of not losing data in a disk failure), and was limited to a single disk volume, but was otherwise complete. We decided that it was more important to get some experience with what we had implemented than to implement the rest of the design.

7.1 Supporting Facilities

To make Alpine usable from Cedar, three additional components were required: a directory

system, an implementation of Cedar open files, and a user interface to some of the server's administrative functions.

A directory system maps the text name for a file into a UniversalFile. Since we wish to replace IFS, we must eventually implement a hierarchical directory system whose functionality includes the commonly-used IFS functions, such as file version numbers, as a subset. As a temporary expedient, we implemented a very simple directory system that provides a flat name space to each file owner. The simple directory system is not suitable for storing a large number of files, as would be required of an IFS replacement, but has been sufficient to support the use of Alpine for database applications.

A Cedar *open file* is an object that implements operations such as Read (pages from file to virtual memory) and Write (pages from virtual memory to file). In Cedar, all but a few low-level clients of files access them via open files, so implementing this abstraction integrates Alpine with Cedar to a considerable degree. For instance, file streams are implemented using open files, so any program that takes a stream as input can be passed a stream for an Alpine file.

A Cedar user sometimes needs to deal with an Alpine server directly, as when checking a disk quota or changing a file access list. Cedar provides the user with an expression interpreter, so, in principle, a user can query and manipulate an Alpine server by making calls to the various Alpine interfaces. This is inconvenient because of the large amount of "boiler-plate" code that it takes to bind to the server, create a transaction, and so forth. So we implemented a set of procedures, analogous to the administrative procedures exported from the Alpine server, such that each procedure call runs as a new transaction. This "Alpine commands package" runs on each user workstation as an ordinary Alpine client, and any Cedar user is free to modify the package as he sees fit.

In addition to these new components, the existing database management system was modified to use Alpine files as well as XDFS files. And to make it sensible for users to try our file system, we implemented a trivial backup system that copied changed files from an Alpine server to an IFS server during off hours.

7.2 Applications

Cypress [Cattell, 1983] is an entity-relationship database management system that evolved from DBTuples [Brown *et al*, 1981]. Cypress now uses only Alpine file storage. Several applications of Cypress have been written, including a system for storing electronic mail messages, a general-purpose database browser, a simple illustrator, and a database containing relationships between Cedar modules.

The message filing system, called Walnut, is by far the most heavily used application of Cypress and Alpine. Walnut represents its data as two files: a Walnut log file and a Cypress database. The Walnut log file contains the messages themselves and a record of all updates, while the Cypress database is used as an index. The Walnut log file can be stored in either a workstation's file system or an Alpine file system.

Because of the various shortcomings of XDFS, discussed in Section 2, it was never practical to store Walnut logs and databases on XDFS. Instead, Walnut used the Cedar workstation's Pilot file

system [Redell *et al*, 1980] which provided transactions. Later, Walnut users were given the option of storing their files on an Alpine server, and almost all of them did. Alpine gave them a noticeable performance improvement over Pilot, but the main reason for its popularity was that it allowed a user to access his filed messages from any Cedar workstation, not just his personal workstation.

Luther, the public Alpine server, has been in service since March 1983. (Luther Pass is located in Alpine County, California.) Since July 1983 the Alpine code running on Luther has not been changed in a significant way, and Luther has averaged about two restarts per week. About 40% of the restarts were due to hardware problems and a microcode bug, and 10% were to install software changes. All but a few of the remaining restarts were caused by three specific Alpine bugs that have been fixed, or the fix is understood. A restart generally took about two minutes; on two occasions, restart failed and the system was cold-started (ignoring the state of the log).

Luther is the first server at PARC to use the Dorado [Clark *et al*, 1981] as its processor. Some flaws that are acceptable when the Dorado is used as a workstation are intolerable when it is used as a server. One two occasions Luther has been unavailable for more than a day due to problems with Dorado hardware. We are now considering a configuration with a warm spare processor to reduce this problem.

Alpine users have several complaints with the system that are unrelated to the hardware. The most often heard complaint is that Alpine aborts transactions too frequently. Clients use the feature of *AlpineTransaction.Finish* that allows them to commit updates but immediately create a new transaction, maintaining the consistency of their application data structures by not releasing locks on Alpine. This works fine when the locks are on a private database, but is a bad idea when the locks are on a shared database because it leads to lock conflict or deadlock. Recall that extending a file requires a write lock on a page of the owner database. This lock is not released by *AlpineTransaction.Finish* with *continue* = TRUE. We are now considering whether to fix this specific problem or to provide a general facility to identify lock requests as being of short or long duration.

There are two other aspects of the too-frequent transaction abort problem. First, Alpine provides the user with no way of finding out why a transaction has aborted. This is clearly a mistake; it will not be hard to log a bit more information about the cause of transaction abort, and let a client read this information from the log.

Second, applications such as Walnut were developed using Pilot transactions in a local file system. These transactions aborted only due to specific request or due to a workstation crash. So applications were not structured with smooth recovery from transaction abort in mind. The next generation of applications will hide the low-level transaction concept from users.

In spite of these deficiencies, Alpine has succeeded in its primary goal: supporting research in database management systems and applications.

7.3 Performance

We have done no performance tuning of Alpine; none of our users has asked for it. We are in

the fortunate position of running the system on a Dorado, which is roughly ten times as fast as an Alto for executing Mesa code. (This factor is not easy to evaluate since Cedar programs tend to use more 32-bit quantities than Alto/Mesa programs did, and the Dorado has 16-bit data paths.) A simple Mesa instruction takes 125 nanoseconds, and a procedure call and return takes 5 microseconds. We designed Alpine to have good performance on slower machines, but have never run it on one. Our Dorado comes with 2M bytes of main store, the minimum configuration, and Luther arbitrarily uses 200K bytes of virtual memory for file buffers.

Prior to writing this paper, we had two rather offhand sources of performance information. First, we noticed the time to copy large files between a workstation and Alpine, using the Copy procedure included in the Alpine commands package. This copy procedure uses a single process that loops calling AlpineFile.ReadPages or WritePages as appropriate, transferring twelve pages in each call. For large files, copying proceeds at 450 Kbps. This includes data transport over a 3 Mbps experimental Ethernet.

Second, we noticed the statistics that are kept by the Cypress database management system. Cypress maintains the average time per call for several internal procedures, including each Alpine procedure that it uses. Cypress uses AlpineFile.ReadPages and WritePages to transfer a single page per call. When called from Walnut, the average time Cypress sees for AlpineFile.ReadPages is varies between 20 and 40 milliseconds, and the average time for WritePages is about 10 milliseconds.

When preparing this paper we ran the file system benchmarks used by Mitchell and Dion in their comparison of XDFS and the Cambridge File Server [Mitchell and Dion, 1981]. Their benchmark uses a single 256K byte file. The first test measures the time taken required for a null remote procedure call, and the second measures the time for a null transaction. The third test measures the average time per read when performing 100 random reads in a single transaction, and the fourth test measures the time per write when performing 100 random writes in a single transaction. The fifth test measures the time to write the entire file sequentially. The results:

1. Remote call, no parameters, no results: 1 millisecond [Birrell and Nelson, 1983]
2. AlpineTransaction.Create; AlpineTransaction.Finish: 10 milliseconds
3. Average of 100 random reads: 25 milliseconds
4. Average of 100 random writes: 8.9 milliseconds
5. Write 256K bytes to existing file (writing 512 bytes per call): 4.8 sec
   Write 256K bytes to existing file (writing 2048 bytes per call): 2.6 sec
   Write 256K bytes to existing file (writing 8192 bytes per call): 3.2 sec

Test 2 should involve no I/O, but does involve a potentially remote call from the coordinator to the worker (the call turns out to be local in the test), and this call is done from a separate process so that multiple workers can be called in parallel (there is only one worker in the test). In tests 3 and 4, the good relative performance of random writes versus random reads demonstrates the advantage of converting random I/O into sequential (log) I/O. Test 5 shows that, given the small pages used in Alpine, it is important to be able to send a run of pages in a single call. The peak performance of 800 Kbps for sequential writing can surely be improved by some work on the Log

component. It may be surprising that writing more than 2048 bytes per call reduces performance; this is probably due to the increased cost in the RPC server stub for allocating and deallocating the large buffer to hold the page run. We should improve this by using a special-purpose storage allocator.

7.4 Effort and Code Size

We first conceived of building Alpine in December 1980. During early 1981 we wrote memos describing the system goals and formed the present group of implementors. At first, none of the implementors was able to devote full time to the project; later, one implementor was. Alpine activity in 1981 was limited to reading published papers on crash recovery and concurrency control in database systems, and writing design documents. By the end of 1981 we had designed public interfaces that are nearly identical to the ones used today, had divided the implementation into components whose interfaces were less well defined, and had done a detailed design of some components. During 1982 the design and coding progressed. In November 1982 we assembled the existing code into a whole and made it run; we had to stub out the Lock and OwnerDB components because implementations did not exist. By mid-January 1983 these components were real and we demonstrated Walnut on Alpine. In February XDFS was decommissioned. In March through June we debugged Alpine while supporting five users; we then let the number of users increase gradually to the present 45 (in February, 1984). Between five and six man-years of effort have been devoted to the system so far.

An Alpine server contains approximately 21,000 lines of Cedar code in 110 modules that were manually written for Alpine. We say that an interface module is *public* if it is used by clients outside of the component, otherwise it is *private*. The manually written code is distributed as follows (the percentages are of code lines):

Public interfaces:	10%, 17 modules
Private interfaces:	16%, 46 modules
Implementations:	74%, 47 modules
Buffer:	14%, 13 modules
Log:	11%, 11 modules
Lock:	6%, 5 modules
Trans:	12%, 21 modules
File:	28%, 30 modules
OwnerDB:	24%, 18 modules
Miscellaneous:	5%, 12 modules

Another 3800 lines in the server were generated by Lupine, and 2000 lines result from instantiating two Cedar packages seven times.

A Cedar workstation accessing Alpine runs 8500 lines of Cedar code for this task. About 4000 lines are manually written code not also used in the server, and 3200 lines are Lupine-generated code not used in the server. The workstation also runs 18000 lines of Cypress code for database access.

During the project we were surprised by the relative difficulty of the OwnerDB component. The present design is a major simplification of our original scheme, yet it is still the second-largest component in Alpine.

A second surprise is the amount of effort required for workstation code. This is not difficult programming, but there is a significant amount of it to do. Workstation code depends upon higher-level Cedar facilities than the server does. Hence, it is inherently less stable, and the cost of maintaining it during the project is higher per line.

Of the server components, File and OwnerDB would be changed if we decided to turn Alpine into a database server that did not support the intermediate abstraction of an unstructured file with transactions. Of course something equivalent to OwnerDB would still be required, but it would presumably be built using a higher-level access method. This would not actually simplify it very much. So even if we eventually decide to build other access methods into Alpine, we have not written much extra code, and in the meantime we were able to use Cypress code with very little modification.

7.5 Status and Observations

Alpine still lacks two-copy logging and incremental backup, and is limited to a single disk volume. A hierarchical directory system and a Pup-FTP server for Alpine have been coded but not yet integrated with the rest of the system.

We have noticed a few features that seem useful and fit in well with Alpine's design. One is the ability to add a *block size* property to each file. The block size is a number of pages, default one, and represents the unit of fine-grained locking on a file. In this way clients that prefer to deal in 2048 byte pages are not penalized by having to set four locks on each page access.

Another easy feature is the ability to perform *dirty reads*, i.e., to read without setting a read lock. The incremental backup system would like to use this feature, and we can imagine other clients.

We have considered the possibility of providing unlogged updates, as in the Cambridge File Server's standard files. We need to evaluate the effectiveness of the high water mark optimization before making a decision.

At present, every transaction writes one log record when the transaction is created, and this sets an ultimate limit to the duration of a transaction: when this part of the log file needs to be reused, the transaction must go away. We should either implement a more elaborate logging scheme, or we should allow special read-only transactions that write no log records, or both.

One of the disadvantages of a system structure with a single virtual address space is that it becomes difficult to circumscribe the resources used by a transaction. So far we have not adopted

any systematic approach to this problem, which means that an errant transaction can crash the system by, for example, opening a huge number of files.

A simple interactive database application tends to hold a transaction open for long periods of time. This would not be necessary if there were an efficient way of validating a workstation's cache of data from the remote server. This should be possible at the level of cached file pages, but no general cache package of this form has been implemented for Alpine.

Sadly for us, the experiments in database access methods that we think are so easy to perform with Alpine have not yet happened, so we cannot report on how difficult or easy these experiments were.

Some sources of unnecessary contention in a database system with physical locking are evident. One is the allocation of free storage in a database. If there is a single free list and it is modified on each allocation, then all allocations are serialized. Perhaps by building a few primitives of this sort into the file model it is possible to reduce contention without building access methods into the server.

8. Using Cedar

The Alpine project is distinct from the Cedar project, but the two projects have close ties. Cedar's database applications need Alpine for database storage; Alpine needs the Cedar language and Cedar nucleus, and was developed using the tools from the Cedar programming environment. As the implementors of an ambitious Cedar application, we are in a good position to comment on the system's strengths and weaknesses.

8.1 Garbage Collection and Finalization

Cedar has a storage management system [Rovner, 1984] that is fully integrated with the language's type system and has automatic reclamation of unused storage, i.e., garbage collection. Cedar's automatic storage management exists at an unusually low level in the system, and it is used by all the main operating system components except for the virtual memory manager. (This structure is not unique to Cedar; Lisp Machine Lisp [Weinreb and Moon, 1978] and Interlisp-D [Teitelman and Masinter, 1981] are built in this way.)

Automatic storage reclamation is obviously a convenience to the programmer, and is considered an absolute necessity in some programming communities. But other programmers consider languages like Bliss, C, and Pascal to be the highest-level languages suitable for "systems programming"; one often hears concerns about the cost or unpredictable performance of garbage collection. Since a file server is clearly a systems programming project, our experience in building Alpine using Cedar's automatic storage management system should be of interest.

Cedar's garbage collector is incremental and operates as a background activity. Without this property, we would not be able to use garbage-collected storage, because we cannot regularly refuse

service to clients for the ten or so seconds it takes a trace-and-sweep garbage collector to do its work. In exchange for the convenience of an incremental garbage collector, we must be careful to avoid circular data structures, or to break up the circularities before releasing such structures. This was not a problem in Alpine.

In implementing Alpine we consciously minimized the allocation of new storage in places where we expected performance to matter most, for example, in the main-line operations for reading and writing individual file pages. Well under ten percent of the code falls into this category. Elsewhere in the implementation we allocate storage as seems convenient; this definitely makes the code easier to write and to understand. But there are qualitative differences even between the code that minimizes allocations and code that we have previously written in standard Mesa, which lacks garbage collection. The differences are known to implementors of production Lisp programs, who also must minimize storage allocation in some situations.

First, all debugging is done at a level that hides the format of objects in memory and the details of the storage allocator. This is possible because essentially all of Alpine is written in a subset of the Cedar language that allows the compiler to check that the program cannot destroy the garbage collector's invariants. Only the call on the Cedar nucleus that reads from the disk into virtual memory buffers cannot be checked by the compiler. In Mesa, dangling references occur and a programmer must sometimes debug at the level of the underlying machine.

A second difference is that cleanup in exceptional condition handling need not be performed so carefully. If a Mesa a program cleans up by freeing some set of allocated objects, the corresponding Cedar program has no explicit cleanup at all.

The most important advantage of automatically managed storage is that it simplifies the design of interfaces between modules and simplifies the design of data structures that are shared between processes. Procedures can accept references to arbitrary data structures or compute and return such objects without concern over whether the caller or the callee "owns" the storage. This is especially important when the procedures are possibly remote.

Sometimes there is an explicit action that must be performed when a collectible object is to be reclaimed. For example, there is a hash table that maps file identifiers to the volatile objects representing files actively in use. When a file object ceases to be referenced from anywhere in the system, we need to remove it from the hash table. To accomplish this we use Cedar's "finalization" mechanism, which permits arbitrary programmer-provided code to be executed when an object is about to be garbage collected, or when a specified number of references to the object remain. In this way, Cedar's automatic storage management facility helps us to take care of a task that would otherwise have to be done by manual reference counting.

While developing Alpine, we identified several deficiencies in Cedar's implementation of automatic storage management. Most notable was a limit on the number of simultaneous references to an object; exceeding this limit would make the object's reference count become "pinned," causing it not to be reclaimed until a trace-and-sweep garbage collection was invoked. Our experience with

Alpine and other Cedar applications led to a significant re-engineering of the Cedar automatic storage management facility to eliminate all the arbitrary limitations that had existed.

8.2 Lightweight Processes

With Alpine, we reaffirmed our belief in the value of lightweight process machinery, a legacy of Mesa that has been carried over into Cedar [Lampson and Redell, 1980]. In Cedar, a system can have hundreds of processes, if necessary. The cost of switching between processes is negligible, and they use the same address space so data sharing is cheap. This leads to a programming style in which separate processes are used for separate activities, however small. The process machinery eliminates the need to explicitly multiplex or schedule activities (except where shared data must be referenced) and improves concurrency.

The remote procedure call mechanism exploits lightweight processes to the fullest. On the server, each incoming procedure call causes a process to come into existence; when the procedure returns, the process terminates. A remote procedure call can be of arbitrary duration, since it executes concurrently with other procedure calls and activities.

8.3 Lupine and Cedar RPC

As previously discussed, the RPC facility enables inter-machine communication to be expressed as ordinary Cedar procedure calls, with the actual data representations and network protocols taken care of automatically. The remote procedure call semantics resemble the local ones very closely, and there are only a few differences that client programmers need be aware of.

While implementing Alpine, two inconvenient aspects of RPC were most noticeable. First, one semantic difference between remote and local calls is that in the remote case the caller and callee do not share the same address space. For Alpine, this means that the caller cannot refer to an object such as an open file or a transaction simply by presenting a pointer to that object, but must instead present some identifier that names the object. The callee must maintain a data structure that maps identifiers to objects, and the caller must manage the set of active identifiers somehow. It seems possible that, in some cases, the RPC system could manage surrogate objects for the client, send the object identifier down the wire, and make the call in terms of the real object at the server.

Second, Cedar provides no convenient way to manage and refer to a dynamically varying collection of instances of the same interface, each located at a different server. This problem seldom arises in the strictly local case, since it is rare for there to be more than one instance of any given interface, and therefore the binding can be specified by a static configuration (for which Cedar's facilities are well developed).

To overcome these inconveniences, we constructed an "Alpine object package" that resides on the client machine and isolates client programs from direct access to Alpine's remote interfaces. This package maintains surrogate objects corresponding to servers, transactions, and open files. A client

can therefore deal with an open file rather than with an [open-file ID, server] pair. Remote procedures are then called by invoking the corresponding operations on the appropriate local objects. The objects, being allocated from automatically managed storage, are reclaimed when no longer in use, and any desirable remote side-effects (e.g., closing open files) are triggered by means of the finalization mechanism mentioned earlier.

The remote procedure call model (in contrast to the message-passing model) potentially permits straightforward on-line debugging of an application program that provokes an error detected during a call to an Alpine server. When the exception (an ordinary Cedar signal) is raised, the server process executing the procedure is suspended (along with the caller's process, which is already suspended waiting for the call to complete); but the rest of the server, including any other call to the same procedure, proceeds unhindered since each remote procedure call is executed by a separate process on the server.

Conceptually, the call stack extends from the debugger on the client machine, through a chain of procedures on the server machine, and then back to the site of the original call on the client machine. Unfortunately, the Cedar debugger presently lacks facilities for tracing the call stack across machine boundaries; so the part of the stack located on the server (including the actual context in which the exception was signalled) is not visible to the debugger. Such a capability would be a useful addition.

Overall we feel that Lupine and RPC were a boon to Alpine. Lupine translates roughly 600 lines of Alpine public interfaces into roughly 7000 lines of bug-free Cedar code that performs well.

Lupine and RPC do not completely solve the problem of protocol evolution, but we think they will help. With such a small user community, we have never attempted to support two versions of the Alpine public interfaces at one time.

It is worth re-emphasizing the point that RPC does not make a remote call look exactly like a local call. RPC does not attempt to hide the fact that the address spaces are different in the remote case, or that the remote call is to a machine that has failures independent of your originating machine. The great thing about RPC is that the remote calls take a familiar form, and that you are not tempted to call in a different way in the local case out of laziness or to make the local call more efficient. RPC allows local calls to look remote, not the other way around.

8.4 Modules

In Section 4 we noted that a log-based transaction implementation can be structured so that most of it does not depend on details of the underlying file system, and is easy to move from one file system to another. We recently tested this hypothesis in the Alpine project by converting the system to use a new Cedar file system that bore no resemblance to the previous file system. Essentially the entire work of the conversion was confined to the Buffer component, as expected. The interface module construct of Cedar (borrowed from Mesa) encourages this sort of clean decomposition of a

large system. We feel that the interface module, including the type checking across module boundaries that it makes possible, is the strongest feature of the Cedar language.

Alpine used the feature of Cedar (and Mesa) interfaces that allows them to be parameterized at compile time; this is analogous to generic package instantiation in Ada. Unfortunately this feature is difficult to use; often a programmer must define a new interface just to transmit a few parameters to the package, and later must deal with several files generated by the compiler in the instantiation process. The Cedar system modeller [Lampson and Schmidt, 1983] is designed to solve these problems. Without the modeller, we found it more convenient to instantiate a small package by copying its source code in a stylized way. This works fine as long as the package is stable.

8.5 Being Part of a Large System-Building Project

The decision to build Alpine on top of the Cedar system (rather than from scratch) was made relatively early in the lifetime of the Cedar project. This had the inevitable consequence of tying Alpine to the evolution and fortunes of Cedar, a complex and ambitious undertaking whose ultimate outcome was not yet completely understood. Was this a good decision?

There is no question that Alpine derived substantial benefit from using Cedar as a base. In particular, the automatic storage management and remote procedure call facilities were sources of leverage, though RPC probably would have been developed anyway. Later, when Cedar reimplemented its virtual memory and file system, we were able to suggest features that made these more usable to Alpine, without building them ourselves.

On the other hand, using Cedar had considerable costs as well. Early on, we spent a great deal of design time considering how to adapt to aspects of Cedar that were less than ideal for Alpine's purposes; this was particularly true of the Pilot file system and virtual memory that Cedar used initially. We experimented with ways to take advantage of new Cedar language features as they became available, such as opaque types and object notation for procedure calls, and these experiments did not contribute much to Alpine. As Cedar evolved, Alpine repeatedly had to be converted to newer versions. However, there were pleasingly few situations in which shortcomings or bugs in the Cedar system held up progress on the Alpine project.

Actually, the most serious diversion of time and effort was due to the Alpine developers also being active participants in the Cedar project. We often decided to pursue activities of ultimate benefit to both Cedar and Alpine rather than focusing exclusively on Alpine. Of course, all this is really a project management issue; and the consequences are a reflection of the informal project organization that is typical of our research environment. Considering both Alpine and Cedar in that context, we are very pleased with the outcome.

Acknowledgments

Many people contributed in one way or another to the Alpine project. Butler Lampson and Forest Baskett gave us some good ideas and encouragement during the early going. Andrew Birrell and Bruce Nelson produced the Cedar Remote Procedure Call facility, which allowed us to concentrate on providing file service while they were providing data communication. The members of the Dorado and Cedar projects helped create a productive programming environment that was the basis for Alpine.

References

[Birrell *et al*, 1982]

Andrew D. Birrell, Roy Levin, Roger M. Needham and Michael D. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM*, **25**, 4, April 1982.

[Birrell, 1984]

Andrew D. Birrell, "Secure Communication Using Remote Procedure Calls," submitted to *ACM Transactions on Computer Systems*.

[Birrell and Nelson, 1983]

Andrew D. Birrell and Bruce Jay Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, **2**, 1, February 1984.

[Boggs *et al* 1980]

David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe, "Pup: An Internetwork Architecture," *IEEE Transactions on Communications*, **Com-28**, 4, April 1980.

[Brown *et al*, 1981]

Mark R. Brown, Roderic G. G. Cattell, and Norihisa Suzuki, "The Cedar DBMS: A Preliminary Report," *Proceedings of ACM-SIGMOD 1981*, April 1981, pp. 205–211.

[Cattell, 1983]

R. G. G. Cattell, "Design and Implementation of a Relationship-Entity-Datum Data Model," Xerox PARC technical report CSL-83-4, May 1983.

[Clark *et al*, 1981]

Douglas W. Clark, Butler W. Lampson, Gene A. McDaniel, Severo M. Ornstein, and Kenneth A. Pier, "The Dorado: A High-Performance Personal Computer," Xerox PARC technical report CSL-81-1, January 1981.

[Gray, 1978]

James Gray, "Notes on Database Operating Systems," *Operating Systems—an Advanced Course*, Springer Lecture Notes in Computer Science, **60**, 1978, pp. 393-481. Also appeared as IBM Research Report RJ2188, February 1978.

[Gray *et al*, 1981]

Jim Gray *et al.*, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, **13**, 2, June 1981, pp. 223–242.

[Israel *et al*, 1978]

Jay E. Israel, James G. Mitchell, and Howard E. Sturgis, "Separating Data From Function in a Distributed File System," Xerox PARC technical report CSL-78-5, September 1978.

[Lampson, 1983]

Butler Lampson, "Hints for Computer System Design," *Proceedings of the Ninth ACM Symposium on Operating System Principles*, October 1983, pp. 33–48.

[Lampson, 1984]

Butler Lampson, "Cedar Language Reference Manual," Xerox PARC internal report.

[Lampson and Redell, 1980]

Butler W. Lampson and David D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM*, **23**, 2, February 1980, pp. 105–117.

[Lampson and Schmidt, 1983]

Butler W. Lampson and Eric E. Schmidt, "Practical Use of a Polymorphic Applicative Language," *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages*, January 1983, pp. 237–255.

[Lindsay *et al*, 1979]

B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, and B. W. Wade, "Notes on Distributed Databases," IBM Research Report RJ2571, July 1979.

[Mitchell and Dion, 1981]

James Mitchell and Jeremy Dion, "A Comparison of Two Network-Based File Servers," *Communications of the ACM* **25**, 4, April 1982, pp. 233–245.

[Mitchell et al, 1979]

James G. Mitchell, William Maybury, and Richard Sweet, "Mesa Language Manual, Version 5.0," Xerox PARC technical report CSL-79-3, April 1979.

[Peterson and Strickland, 1983]

R. J. Peterson and J. P. Strickland, "Log Write-Ahead Protocols and IMS/VS Logging," *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1983, pp. 216–243.

[Redell *et al*, 1980]

David D. Redell *et al.*, "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, **23**, 2, February 1980, pp. 81–92.

[Rovner, 1984]

Paul Rovner, "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language," to appear.

[Svobodova, 1984]

Liba Svobodova, "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys*, to appear. A draft appeared as an IBM Research Report.

[Teitelman, 1984]

Warren Teitelman, "The Cedar Programming Environment: A Midterm Report and Examination," Xerox PARC technical report CSL-83-11, June 1984.

[Teitelman and Masinter, 1981]

Warren Teitelman and Larry Masinter, "The Interlisp Programming Environment," *Computer* **14**, 4, April 1981, pp. 25–34.

[Weinreb and Moon, 1978]

Daniel Weinreb and David Moon, "Lisp Machine Manual," MIT Artificial Intelligence Laboratory, November 1978.

XEROX

The Alpine File System

Mark R. Brown, Karen Kolling, and Edward A. Taft

CSL-84-4