The Alpine File System

MARK R. BROWN, KAREN N. KOLLING, and EDWARD A. TAFT
Xerox Corporation

Alpine is a file system that supports atomic transactions and is designed to operate as a service on a computer network. 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. We report on using the Cedar language and programming environment to develop Alpine.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; *file organization*; D.4.5 [**Operating Systems**]: Reliability-*backup procedures*; *checkpoint/restart*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Design, Experimentation, Reliability

Additional Key Words and Phrases: File servers, atomic update, recoverable files, remote procedure calls

1. INTRODUCTION

Alpine is a file system that supports atomic transactions and is designed to operate as a service on a computer network. (The system is known within Xerox as "Research Alpine," sometimes abbreviated "RAlpine." For simplicity, we call it "Alpine" in this paper.)

A comprehensive survey article [26] describes the concept of atomic transactions and the notion of file service on a computer network. 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.

At the time this paper was written (February 1984), an Alpine server stored the personal and shared databases used daily by about 45 people at Xerox PARC.

Authors' addresses: M. R. Brown and K. N. Kolling, Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301; E. A. Taft, Adobe Systems Inc., 1870 Embarcadero Rd., Suite 100, Palo Alto, CA 94303.

It had been available as a service for ten months. The system did not meet all of its goals, and was still being developed.

Section 2 of this paper describes the situation that led us to build Alpine: we needed it in order to continue doing research on databases and applications. 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 described in Section 2.

Section 4 discusses the two 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 had a large impact on the rest of the design. In Section 5 we present that design in detail, explaining the 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, present some information about Alpine's performance, and give the current status and plans for the system.

Alpine is written in the Cedar language [15], using the Cedar programming environment [25, 27]. Section 8 describes the effect that using Cedar had on the development of Alpine. We defer most discussion of Cedar to Section 8, but a few facts will help with the intervening sections. The Cedar language evolved from and is similar to Mesa [10, 20], 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 nucleus is a basic environment for running Cedar programs. It includes a paged virtual memory and a simple file system with a read-write interface. The Cedar nucleus does not provide either separate address spaces or a protection boundary between the nucleus and its clients; the Cedar language's type checking makes it very unlikely that a nonmalicious program will corrupt another program or the nucleus.

2. MOTIVATION FOR BUILDING ALPINE

Among the research topics being investigated in Xerox PARC's Computer Science Laboratory was the use of database management systems to support work in office applications and programming environments. In our laboratory, all shared files, including files that represent databases, were stored on file servers. Therefore, research involving shared databases required an appropriate file server.

Xerox Distributed File System (XDFS), also known as Juniper [14, 19], was a file server implemented on an Alto and designed to meet this goal. XDFS supported random access to files and provided atomic transactions. We constructed a database management system [6] and several applications that used XDFS to store shared databases. But the database applications were not usable because of problems with XDFS. The basic operations such as creating a transaction, reading and writing file pages, and committing a transaction were

slow but tolerable. A larger problem was that the server crashed frequently, and recovery from any crash took over an hour. The largest problem was that only a few dozen randomly distributed file pages could be updated in a single transaction; the Alto's small memory soon filled up with data structures related to the updates. At that time, the Cedar programming environment project was well underway and XDFS, written in Mesa for the Alto, was an unattractive basis for further development.

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. IFS (interim file server) stored, and still does store, nearly all of our laboratory's shared files. IFS was written in Bcpl for the Alto in 1977 and was designed for transferring entire files to and from a personal workstation's local file system. It does not support random access to files, so there was no question of using it to store databases.

One reason for replacing IFS was 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, we planned to include a local file system that supported transparent caching of remote files in Cedar [24]. 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. Extensions to IFS in support of transparent file caching would be desirable, but IFS would not be easy to modify.

For these reasons, we hoped that Alpine could replace IFS as well as 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. Many of the problems with XDFS can be traced back to the Alto's limitations. 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. Cedar's large virtual and real memory are what allowed Alpine's goals to be more ambitious than those of IFS.

A typical database system implements a set of access methods as a layer on top of a file system and implements transactions as part of each access method. Access methods are disk-based data structures like B-trees and hash tables. To provide transactions, an access method typically sets locks and writes log records. Instead, we have a database system on top of a transaction-based file system. Why not go the other route and build a typical database system? There was one clear reason not to base a database system on file-level transactions, but there were four reasons for hoping that our decomposition would work anyway.

The clear disadvantage of file-level transactions is lower performance, caused by the file system's inability to use database-level semantics to optimize concurrency control and recovery. The file system locks files or pages; a database system

that performs its own concurrency control can set locks on logical units and avoid some artificial conflicts that arise from file or page locking. Similarly, the file system may have to write more bytes to disk to make a page write recoverable than a database system does to make a higher-level operation (such as a B-tree insertion recoverable).

Though this loss of performance would clearly not be acceptable in a system designed for high-speed processing of banking transactions, the trade-off may be different in the application areas on which we concentrate: office applications and programming environments. It seems likely to us that the performance lost to extra lock conflicts and extra logging can be recouped by developing new access methods, avoiding bottlenecks at the server processor, and escaping from the database system when that seems appropriate.

The first advantage of file-level transactions is that they simplify the client database management system that builds on them. 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 to write a single page to disk.

Simplifying the database system is especially important if the database system is itself an object of research. Database systems that were originally designed for business applications are now being used 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 existing database systems has hindered experimentation with them.

The second advantage of file-level transactions is that they provide the option of running the database system on a separate machine from the file system. This moves some processing load away from a machine that is accessed 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 closely coupled database servers.

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 not secure, and hence it can provide ironclad access control only at the level of files, not at the finer granularity of database record sets or fields. In our laboratory, neither of these factors rules out locating the file system and database system on different machines.

The third advantage of file-level transactions is that they ensure that an application can use the same transaction abstraction to deal consistently with all data stored in the file system. An 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.

In addition to these potential advantages, there was also the immediate advantage that we already had a database management system that assumed a transaction-based file system. We were prepared for the possibility that even after replacing XDFS we would find that this architecture would not meet our needs. But in that case, not too much effort would have been wasted as long as most of the implementation of file-level transactions carried over to the database system.

3. GOALS

The purpose of Alpine was to support other research projects; Alpine was not an end in itself. This had some noticeable effects. It encouraged conservative design—where possible, we wished to use proven techniques from the published literature and even existing code. (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 would allow the system to begin supporting its primary clients, database systems, without requiring the entire file system to be complete. We have much to learn about system support for databases, and people here who write database applications have much to learn about using shared, remote databases. The sooner we began 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 and 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 builder of a file server must make decisions on 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 had already decided that Alpine should implement atomic transactions on files but we wanted it to allow concurrent update of a file by several transactions. Alpine's reliability goal was to support, but not require, configurations that survive any single failure of the disk storage medium without loss of committed information.

Availability. Our environment did not require that a file server provide continuous availability; we could tolerate scheduled downtime during off hours and small amounts of downtime due to crashes during working hours. 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 seemed likely to us that setting higher goals for availability would have a high cost and little payoff for our clients.

File access. We wanted Alpine to 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 since it is no great advantage to provide access to arbitrary byte sequences. The goal was 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 a single transaction should be able to include files from several different Alpine servers. This seemed useful and did

not present any large implementation problems. Adequate techniques for coordinating multimachine transactions are understood [18], and such transactions are supported by commercial systems including IBM's CICS [13] and Tandem's TMF [5].

Capacity. Our large IFS servers typically controlled six disk drives. We thought that Alpine would need to support at least this many drives.

Access control. We decided that Alpine should implement a simple access control scheme with a read and a modify access list for each file. This is all that a file system needs to provide a database management system. A database management system that wants to enforce complex controls over access to data will need to implement them itself, since they will depend on the format of the database. The modify access lists of an Alpine file controlled by such a system won't name people, only database servers, thereby forcing all accesses to the file to go through one of the servers.

Based on our experience with IFS, we think that in our environment, a file controlled by a simple database management system or an ordinary file needs nothing more elaborate in the way of access control.

Accounting. We decided that Alpine should enforce quotas over the allocation of disk space to various users and projects. After all, IFS provided such quotas, and IFS was a usable system. Using the file server to store databases does not create new problems in this area. A file server without space quotas is not very usable, even in our research environment.

In addition, we had to consider whether Alpine should be designed to run on a Cedar workstation as well as a server. This option is sensible because a Cedar workstation is a powerful computer with its own rigid disk. Such a single-disk system could not survive all single storage medium failures without a loss of committed data, but can still be useful. For instance, running Alpine on a workstation could support local database storage and small system configurations without a dedicated Alpine server.

Therefore, it seemed worthwhile to provide the option of running Alpine on a workstation as long as the development cost was sufficiently low. To keep the development cost down, we continued to optimize the design for the multidisk server case, doing nothing extra to adapt it to the single-disk case. We also imposed the restriction (already assumed in the server design) that Alpine can never be the only file system on a computer; a standard Cedar file system must always be present for storing boot files, virtual memory backing files, and such. (The two file systems can share a single disk drive using Cedar nucleus facilities [24].) As a result, providing the workstation configuration adds almost no code to Alpine's implementation.

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

(a) A directory system (mapping the text name of a file into its internal file system name) would be required in order to make Alpine usable, but a directory system capable of looking up a few files representing databases would be much less work to implement than a directory system that emulates IFS. In the future we may aspire to include a distributed directory system with more location

transparency and some form of file replication. So there are advantages in avoiding a strong coupling between Alpine and its directory system.

(b) A Pup-FTP server [4] would be a client of the directory system and of Alpine. The FTP server was not required for database access, but was necessary to replace IFS.

(c) A system for archiving files from the primary disk storage of a file server and for bringing back archived files would be a client of the directory system and of Alpine. (IFS does not have an archiving system.)

4. TWO MAJOR DECISIONS

Two decisions helped us focus our attention on our primary task—providing transactions on files for use by a database system. Using the log technique to implement atomic file update allowed us to stay out of the business of building the lowest levels of a file system; using RPC allowed us to stay out of the communications business. Merely saving implementation effort might have been enough to justify these decisions, but in fact, we felt, even then, that they would lead to a better-performing system than anything else we could conceive.

4.1 Implement Atomic File Update Using a Log

The two techniques most often discussed for implementing transactions on files are logs and shadow pages. One published paper [12] argues that logs are better than shadow pages for large shared databases, and many database systems use logs. On the other hand, a recent survey of transaction-based file servers showed that, except for Alpine, they have all chosen the shadow page technique [26]. What are the tradeoffs between shadow pages and logs? Why did we choose the log technique for the Alpine file server?

To describe the shadow page technique, we must clarify some terminology concerning files in general. A file *page* is 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 file is named by a *file ID* and a page within a file is named by a *page number*. A disk *sector* is a region of a disk that is capable of storing the bytes of a page. A file system implements files using disk sectors.

The *file map* is a representation of the mapping from the pair [file ID, page number] to disk sector. To read a file page, the client presents a pair consisting of file ID and page number; the file system uses the file map to find the right disk sector and returns its contents. The *allocation map* gives the free or allocated status for every disk sector.

The shadow page technique updates a file by updating the file map. The file system allocates a new sector, writes the new page value there, and modifies the file map so that reads go to the new sector rather than to the old one. If the client commits the transaction, the change to the file map is made permanent, and the old sector is freed. If the client aborts the transaction, then the file map is restored to its previous state and the new sector is freed. After a file page has been written and before the change is committed, the page exists in two versions: the new but uncommitted page and the old page, which must be preserved until the update is committed. The old page is called the shadow page [12].

There are many variations of the shadow page idea. They largely concern the management of the file map and the allocation map. Various representations are possible, and for each there are many different algorithms that can be used for updating in a way that is consistent with the outcome of transactions.

To have transactions on files, it is necessary to keep different versions of pages around. In the shadow page technique, the versions are stored as pages that are incorporated into files by modifying the file map. In the log technique, they are stored in log records and are incorporated into files only by copying. In the log technique, a file that allows update under a transaction is represented as the log plus a file in an underlying file system that does not implement transactions.

There are several different log techniques; ours is based on *redo logs*. A log record for page update contains the transaction ID of the transaction making the update, the pair [file ID, page number] of the page being updated, and the entire new value of the page. The file system does not write the new value of the page to the underlying file before the transaction commits. It writes a page update log record and makes an entry in a data structure called the *volatile file map*. The volatile file map contains entries that map a pair [file ID, page number] to a log record.

The file system starts a read by consulting the volatile file map. If there's an entry for this [file ID, page number] pair, the file system reads from the log; otherwise, it reads from the underlying file. The presence of a volatile file map entry indicates that the current value of the page resides in the given log record; the absence of an entry indicates that the value is in the underlying file.

To commit a transaction, the system does not have to make the underlying file reflect all the updates made by the transaction. It writes a commit log record and then waits only until the commit record and all other log records written by the transaction have reached the disk. (Something analogous to the commit record is also required in the shadow page technique.) The system then creates a background process. For each volatile file map entry made by the transaction the process copies the value out of the log and erases the entry. The actual disk write to the underlying file is performed asynchronously by the buffer manager, possibly long after the transaction's background process has finished its work. To abort a transaction, the system just erases the volatile file map entries it has made.

The log is represented as a fixed-length file in the underlying file system, divided into variable-length log records. To keep things simple, the log records are allocated in a queue-like fashion. The storage for the oldest log record can be reused when its transaction has committed, the value has been copied out of the log, and the actual disk write has been made to the underlying file. (The storage can also be reused if the transaction aborts.) The log implementation includes a checkpoint process to ensure that disk writes eventually reach the underlying file.

The differences between shadow pages and logs can be illustrated by considering a client that writes ten pages to an existing file under a transaction and commits the transaction. Compare the I/O activity:

Ten times, the shadow page implementation allocates a disk sector and writes a new page value to it. It also writes enough additional information to disk so that the file and allocation maps can remain correct regardless of whether the transaction commits or aborts. At commit time it writes a commit record of some

sort onto disk. After the commit, the ten sectors containing the current pages are incorporated into the file map and the ten sectors containing the shadow pages are freed.

Ten times, the log implementation writes a log record slightly larger than a file page. At commit time, it writes a commit record and forces the log to disk. After the commit, it copies data out of the log to buffers associated with the underlying file. The buffer manager eventually writes the dirty buffers to disk.

Which technique is preferable? The shadow page implementation seems to require less I/O, because each page is written only 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 nonvolatile file map, which is not necessary in the log implementation, but in theory this should require fewer than ten page writes because many data pages are referenced by each file map page. We, nonetheless, find the log technique preferable. It has five advantages: three performance advantages, one added function it supports, and one implementation advantage.

The first performance advantage is that the total cost of the I/Os performed in the log technique is often less than in shadow pages. Both techniques sooner or later write the new page onto a file page on the disk; both have to write commit records. There the similarity ends. The shadow page technique must also recoverably update the file map and recoverably update the allocation map, both to allocate disk sectors and to free them. The log technique must also write log records and read them. Though the total number of pages touched by the shadow page technique is sometimes smaller, every read and every write is random and therefore expensive. In the log technique, the writing is always sequential and therefore cheap, and the reading is often buffered and therefore free.

In the shadow page scheme, the file map is updated by every transaction, whereas in the log scheme it's updated only when the number of pages in a file changes. The theory in shadow page technique was that a page of file map would cover several pages updated by the same transaction; but in fact, as the files get larger this is less and less likely to be true.

Almost every page update in a shadow page system requires the services of the general disk allocator for file pages. In contrast, log records are allocated sequentially from a bounded file, so allocation involves a pointer increment and test. Writes to the log are strictly sequential because we implement the log as a single file. Furthermore, the techniques for writing log records efficiently are well understood [21]. Reads directed to the log will often find the data in the buffer pool because the typical client commits a transaction shortly after performing all of its writes. So if all goes well, each page updated by a transaction 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 second performance advantage of the log technique 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 can improve performance in both sequential and random access. Sequential access goes faster on contiguous files; the extent-based file starts and remains contiguous, while the shadow page file loses its contiguity when it is updated. Random access (as well as sequential access) goes faster when

the file map is smaller because file reads seldom require extra I/Os to access the file map. The file map in an extent-based file system typically needs to represent the location of a few long runs of pages; in a shadow page file system it typically needs to represent the location of each individual page. Therefore, the file map in an extent-based file system is smaller than in a shadow page system.

The third performance advantage is that the log implementation (at least the redo log we have described) defers more work until after commit than shadow pages do. The shadow page technique writes data to file pages before commit, while the redo log technique waits until after. If the system is lightly loaded, or heavily loaded but with a bursty pattern of update requests, deferring work makes update transactions respond faster than doing it before commit. 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, each write to a frequently updated file page need not result in a corresponding disk write. Unlike log writes, which are forced to disk at commit time, file writes take place asynchronously, caused by page replacements and by the checkpoint process of the log manager.

The added function that the log technique supports is file backup. A log-based system needs to do only three things in order to survive any single storage medium failure without loss of committed data: It must maintain two copies of the log file, each copy on a different disk drive, it must periodically take a volume-by-volume dump of the file system, and it must perform *log archiving* by saving one copy of each log record written after the most recent dump has been started. The performance cost of two-copy logging is quite small if the two drives have separate controllers and the drives are dedicated to logging; it is larger if they share a controller or are shared with normal files. Dumping can proceed concurrently with file update activity and hence be nondisruptive [11].

In contrast, the entire idea of the shadow page technique is to write new data just once. With a shadow page system, the only practical backup method short of maintaining a duplicate on-line file system is to take periodic volume dumps and maintain . . . a log. Backup is a larger incremental addition to shadow pages than to logs.

The implementation advantage of the log-based technique is that it can be layered on an existing file system. 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 performance of the log-based implementation. The log-based file system can be structured so that most of it does not depend on details of the underlying file system and can be easily ported from one file system to another. Adding transaction logging to an existing file system is much less work than writing a new file system, with all of its associated utility programs. Given our experience with IFS and XDFS, and our pragmatic goals for Alpine, this advantage seemed very powerful to us.

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 [4]. In principle, client programs viewed interactions with XDFS 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 creating processes. These processes are quite inexpensive [16]. Therefore, 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 XDFS project was well under way. There was no clean separation between the file system and data communications components of XDFS. For instance, a data type defining a 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 decided to use RPC instead. We were quite fortunate that a project to produce a general-purpose remote procedure call facility [3] 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. Lupine imposes some restrictions on interface design; it will not produce stubs to transmit arbitrary data structures from one machine to another.

Thus the functionality of Alpine is defined by a small set of Cedar interfaces, *not* by a network protocol. The full functionality of Alpine is available remotely by making remote calls to these interfaces. The interfaces are also accessible to local clients like a directory system and an FTP implementation.

A general-purpose RPC is not necessarily less efficient than a protocol that is specialized to the task at hand. 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 actually be less than for a more specialized protocol.

5. INTERFACES

In the interfaces that follow, the 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 "}".

But to make sense of the interfaces, some knowledge of Cedar RPC is necessary. The Cedar RPC mechanism uses Grapevine *RNames* for individuals and groups

[1]. An RName for an individual (person or server) looks like "Schroeder.pa"; for a group, "CSL↑.pa".

When a program calls a procedure P in a remote interface I, the server that gets called is determined by the current RPC binding on I. The binding is made by calling the RPC machinery and passing it the RName of the desired server. This means that one need not pass the RName of the server on every procedure call. To get any understanding of how RPC binding works, read Birrell and Nelson [3].

A client must call the RPC system to establish a *conversation* with a server before it can make authenticated calls to the server [2]. 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. To make the procedure declarations easier to read, we omit the conversation parameter.

An Alpine file system exports four public interfaces: AlpineTransaction, AlpineVolume, AlpineFile, and AlpineOwner.

5.1 AlpineTransaction Interface

The AlpineTransaction interface provides two logically distinct abstractions, the transaction *coordinator* and the transaction *worker*. The coordinator manufactures transaction IDs and controls workers in the two-phase commit protocol. The worker performs data manipulation under a transaction and obeys the coordinator during a commit. It is not strictly necessary to expose these distinct services to the client of transactions; XDFS did not. But exposing them makes the AlpineTransaction interface more straightforward to implement than the XDFS transaction interface, and Alpine no harder for our typical client to use.

A transaction ID contains enough unpredictable bits so that it is extremely difficult to forge. Therefore, an Alpine server can treat it as a capability. Any client that has a transaction's ID and the transaction coordinator's RName 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. Raises *OperationFailed*[*busy*] if the server is overloaded. If *createLocalWorker*, the coordinator calls *CreateWorker*[*transID*, RName-of-this-server].

CreateWorker [transID, coordinator: RName]

! Unknown {coordinator, transID}, OperationFailed {busy}.

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*. Raises *OperationFailed*[*busy*] if the server is over-

loaded. 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. 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.

5.2 AlpineVolume Interface

There are several reasons for providing the *volume* as an abstraction of a physical disk volume. A volume often corresponds to a disk arm, so a client can improve performance by locating its most frequently accessed files under different arms. A volume can be dismountable and belong to a single user. A volume can be the unit of scavenging, so dividing a large file system into volumes reduces the delay in restart if a single volume needs to be scavenged. Alpine has no ambitions of its own for volumes; it uses whatever volume structure is provided by the underlying file system. Alpine does assume that volume IDs are globally unique.

Though volumes are useful, they can also be a nuisance. When a user creates a file on a server, he usually does not care which volume it is on. Therefore, every Alpine volume belongs to a *volume group*. The file creation procedure (*AlpineFile.Create*) accepts either a volume or a volume group as a parameter; when a volume group is given, Alpine selects some volume in the group and creates the file there.

A file *owner* is an entity identified by the RName of a person or a group. Each volume group contains an *owner database*, a set of records indexed by the RNames

of valid owners for files in the volume group. Among other things, an owner database record contains the disk space quota for an owner, so that users do not have to manage quotas on a per volume basis. The *AlpineOwner* interface below contains operations on owner databases.

Any procedure that takes a *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 do 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.

5.3 AlpineFile Interface

A *file* is a sequence of 512-byte pages and a set of property values. A file is named by a file ID that is unique within the volume containing the file. A *UniversalFile* is a globally unique file name: a [volume ID, file ID] pair. The pages of a file are indexed starting from zero. The *size* of a file is the number of pages that it contains.

The properties of an Alpine file are given by the enumerated type *Property*, and are the *byteLength*, *createdTime*, *textName*, *owner*, *readAccessList*, *modifyAccessList*, and *highWaterMark*. This set of properties is not expandable by the client. 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.

An Alpine user may read a file if he is either in the file's read or modify access list and may modify a file if he is in the file's modify access list. Each list can contain both RNames and the two special names "Owner" and "World". An RName can name a group, so a short access list such as ("CSL↑.pa", "ISL↑.pa") can name a large number of individuals. Alpine sets a fixed upper bound on the total number of characters in the two lists; the bound is large enough to accommodate more than 20 typical RNames.

A file may contain pages at the end whose contents are considered undefined by Alpine; the high water mark is the page number of the first page with undefined contents. Alpine can optimize writes to undefined pages. When a file is created all of its pages are undefined, and when a file is extended the additional pages are undefined. Alpine advances the high water mark automatically when 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. We expect the high water mark feature to be used in IFS style file transfers.

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 units 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 locks to control concurrent access to files from different transactions. The locks are similar to System R locks [12], with a few differences. The locking hierarchy has two levels only: whole-file locks and page locks. When page locks are used on a file, the file properties are locked as a unit. As in System R, Alpine uses intention lock modes to control the interactions between clients who lock at different levels of the hierarchy. In addition to read and write lock modes (called share and exclusive by System R), Alpine supports an update mode similar to that used in XDFS. A client can lock in update mode and perform uncommitted updates without waiting for existing readers to finish. New readers are blocked by the update lock. If the existing readers are not gone by the time the updater commits, the updater must wait for them to finish.

A client chooses between whole-file locking and page locking when opening a file. If a client chooses page locking, then its page reads and writes implicitly set additional locks. In case of a lock conflict, these procedures wait until either the lock is acquired or the transaction is aborted. A client who is satisfied with this locking behavior need not be concerned further with locks. But more ambitious clients may (1) set locks in order to reserve resources early in a transaction, (2) set stronger locks than the default when performing operations that implicitly set locks, (3) release read locks before the end of a transaction to reduce conflicts, and (4) cause procedures to fail immediately on lock conflict rather than waiting.

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. Alpine is perfectly willing to execute, say, seven *ReadPages* and two *WritePages* calls at the same time for a single 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. If a client attempts to commit a transaction while another Alpine operation is in progress for the transaction, Alpine 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 transaction *transID*. The *volumeID* parameter 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 (a field of an owner database record), 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 contains an identifier which is permanently unique, relative to the containing volume. It may also contain information used to find the file within the volume; this location information is treated as a hint [15], and may change during the lifetime of a 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 reopen 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* for 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, the RPC stub generator, 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].

Sets locks on the specified pages of the file.

UnlockPages [openFileID, pageRun].

If the specified pages of the file are locked in *read* mode, 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 file properties.

SetSize [openFileID, new PageCount, 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 file owner's create access list, 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 new owner's create access list; the disk space occupied by the file is credited to the old owner and charged to the new one.

If the number of characters in the resulting access lists is too large, the exception *OperationFailed*[*insufficientSpace*] is raised. If multiple properties are written by one call and this error occurs, some of the properties may nevertheless have been written successfully.

5.4 AlpineOwner Interface

As previously mentioned, each volume group contains an *owner database*, a set of records indexed by the RNames of valid owners for files in the volume group. The entries in an owner database record are given by the enumerated type *OwnerProperty*, and are the *spaceInUse*, *quota*, *rootFile*, *createAccessList*, and *modifyAccessList*. This set of entries is not expandable by the client.

An owner's space in use is the total number of pages in all owned files, plus a fixed overhead charge per owned file. Hence, space in use is changed as a side effect of *AlpineFile.Create*, *Delete*, *SetSize*, and *WriteProperties* calls. An owner's quota is an upper bound on space in use, enforced by *AlpineFile.Create*, *SetSize*, and *WriteProperties* calls. The owner root file is a *UniversalFile* that can be read and written by clients. It is intended for use by a directory system.

An owner's create access list governs file creation for an owner and access list modification for an owner's existing files. An owner's modify access list controls updates to the owner's create access list; this provides 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 by *AlpineTransaction.AssertAlpineWheel*. We omit the procedures that perform these administrative updates.

ReadProperties [transID, volume GroupID, 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: Buffer, Log, Lock, Trans, File, and OwnerDB.

Buffer provides a buffer manager [11] on the underlying file system. Operations in this 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.

Log provides a log abstraction and controls crash recovery. Log writing and reading is built upon Buffer in a straightforward way. Log writing requires the ability to write a specified page run to disk synchronously, which Buffer provides.

The log checkpoint process is responsible for freeing up log space for reuse. For simplicity, it does this on the basis of transactions rather than of individual log records. The log space used by a transaction may be reused when all of the transaction's updates have been performed at the Buffer interface, and these updates have been written to disk. Therefore, the checkpoint process periodically examines all transactions to determine which of them have completed their updates at the Buffer interface, then calls a Buffer procedure that synchronously writes all currently dirty buffer pages to disk. This procedure is allowed to take a relatively long time and does not interfere with other Buffer operations that may dirty more pages. When this procedure returns, the checkpoint process can reclaim the space used by these transactions. It writes a checkpoint log record to define the start of the log for recovery processing, and writes a pointer to this checkpoint record in a known location to avoid a linear search of the log during restart [12].

The log watchdog process ensures that the checkpoint process makes progress. If the oldest transaction in the log is sufficiently old (in terms of log written since it started), then the log watchdog aborts it.

Lock implements a lock manager [11]. It detects deadlocks within a single file system and aborts transactions to break these deadlocks. Lock also uses timeouts as follows: If one transaction is being blocked by another that has not called the Alpine server lately, then Lock aborts the inactive-looking transaction.

Trans implements two-phase commitment of transactions [18]. Interserver communication for coordinating transactions is performed using RPCs through a private interface not described in Section 5. 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. Its implementation of atomic file update generally follows the redo log scheme discussed in Section 4.1. Elhardt and Bayer have proposed a similar logging scheme for use in implementing a database management system [9]. Their technique writes no log records for a transaction until phase one of transaction commit, and requires that all log records for a single transaction be contiguous in the log. Their technique only reads the log during recovery, but this limits the number of pages written in a transaction to the size of the buffer pool, which resides in main store.

File represents a file's properties using a single leader page. 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 *AlpineFile.Create* and *AlpineFile.SetSize* (when extending the 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. Very little information needs to be written into the log in order to make a disk allocation undoable, which is not the case for a file page write.

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. Alpine forces these writes to disk before a transaction commits.

Because the File component is designed to defer work during normal operation, it falls out naturally for it to defer work during crash recovery as well. That is, the Alpine system can export its public interfaces without bringing the underlying files to a transaction-consistent state.

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. Grapevine performs membership tests for RNames that name groups. The efficiency of this is acceptable because OwnerDB caches the results of calls to Grapevine. OwnerDB discards old cache entries in order to track Grapevine updates; this simple technique suffices because Grapevine's own consistency guarantees are not strong.

7. EXPERIENCE AND EVALUATION

By early 1983, we had coded the implementation and started to test it. We had fallen short of two of our goals: The implementation did not include two-copy logging, volume dumping, and log archiving; therefore, it did not meet the goal of surviving all single storage medium failures without the loss of committed information. The implementation provided a single volume group containing a single volume; therefore, it did not meet the goal of supporting multiple disk drives. Though we were committed to both of these goals, we decided that it was more important to get some experience with what we had built than to build more.

Even without meeting all of our goals, we were able to make progress again in the area of database experimentation. People in the laboratory who want to do database work are no longer being stalled by the absence of a file server that can support databases. We did not replace IFS.

7.1 Supporting Facilities

To make Alpine usable from Cedar, we built three external components: a directory system, an implementation of Cedar open files, and a user interface to the server's administrative functions. Each of these components runs on an Alpine user's workstation as an ordinary Alpine client.

A directory system maps the text name for a file into a *UniversalFile.* To replace IFS, Alpine needs a hierarchical directory system that supports file version numbers and various other features. As a temporary expedient, we implemented a directory system that provides a flat name space to each file owner. This interim directory system is adequate to support Alpine's use as a database repository.

A Cedar *open file* is an object that implements operations such as Read (pages from open file to virtual memory) and Write (pages from virtual memory to open 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 may sometimes need to deal directly with an Alpine server, as when checking a disk quota or changing a file access list. Cedar's expression interpreter allows a user to do this by making calls to procedures in the various Alpine interfaces. But this is inconvenient because of the large amount of "boilerplate" involved in binding to a server, creating a transaction, and so forth. Therefore, we implemented a set of procedures that are analogous to the administrative procedures exported from the Alpine server except that each procedure call runs as a new transaction. Because this "Alpine commands package" runs on a user's workstation, any Cedar user is free to modify it as he or she sees fit.

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

7.2 Applications

Alpine solves the problem that we built it to solve: It supports research in database applications. Cypress [7] is an entity-relationship database management system that evolved from DBTuples [6]. Cypress now uses only Alpine file storage. Xerox researchers have written several Cypress applications, including a system for storing electronic mail messages, a general-purpose database browser, an 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 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 Cedar workstation's file system or an Alpine file system.

Because of the 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 [22] which provided transactions. Later, Walnut users were given the option of storing their files on an Alpine server and almost all did. Alpine provided a noticeable performance improvement over Pilot, but the main reason for its popularity was that it allowed a user to access his or her filed messages from any Cedar workstation, not just his or her personal workstation.

The first public Alpine server, Luther, 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 any significant way, and Luther has averaged about two restarts per week. About 40 percent of the restarts were due to hardware problems and a microcode bug, and 10 percent 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 for which 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 [8] 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. On 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.

Apart from the disruption caused by hardware-related downtime, the complaint heard most often from Alpine users is that Alpine aborts transactions too frequently. Some changes can be made in Alpine's implementation to reduce this; these changes will be discussed in Section 7.5. A large part of the problem is that Walnut and other applications were developed using Pilot transactions in a local file system. These transactions aborted only as a result of a specific request or a workstation crash. So applications were not structured with smooth recovery from transaction abort in mind. The next generation of applications will be designed to hide the low-level transaction concept from users.

7.3 Effort and Code Size

We first conceived of building Alpine in December 1980. During the early part of 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 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 internal 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 gradually increase to its 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):

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

Another 3800 lines in the server were generated by Lupine, the Cedar RPC stub generator, and 2000 lines resulted 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 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 implementation is much simpler than the one we designed originally, yet OwnerDB is still the second-largest component in Alpine.

A second surprise was the amount of effort required for workstation code. This is not difficult code to write, but there is a significant amount of it. In addition, the cost per line of maintaining workstation code is higher than for code in the server. This is because the Alpine server depends on only the most stable Cedar interfaces (the Cedar nucleus facilities for virtual memory and file access), whereas the workstation code depends on higher level interfaces; hence the workstation code is inherently less stable.

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 be built using a higher-level access method. This would not actually simplify it much. Even if we eventually decide to build other access methods into Alpine, we have not written much extra code, and in the meantime we are able to use Cypress with very little modification.

7.4 Performance

The Alpine server in public use at PARC runs on a Dorado processor. Some information on the Dorado's performance follows. When a Dorado runs a Cedar program, the simplest machine instruction takes 125 ns, and a general procedure call that passes no arguments and returns no results takes 9 μs. A Dorado running a Cedar program is about 8 times as fast as an Alto running the corresponding Mesa program. This speed-up varies with a number of factors; we assume that the Alto's display is turned off to reduce memory interference, and that the Dorado's Cedar program deals in 32-bit quantities while the Alto's Mesa program deals in 16-bit quantities. The Alpine server has 2 mbytes of main store, and uses 200 kbytes of virtual memory for file buffers.

The server's I/O devices can be characterized as follows. The Alpine volume lives on a 300-mbyte removable pack disk drive. This drive has a track-to-track seek time of 6 ms, a maximum seek time of 55 ms, and an average seek time of 30 ms. The drive has an average rotational latency of 8.3 ms and a transfer rate of 1.2 mbytes/s. The server is interfaced to a 2.94 mbits/s experimental Ethernet.

We were most interested in Alpine's performance when running database applications, given that we built Alpine to support them. The Cypress database management system measures the average elapsed time per call for several of its internal procedures, including all file I/O procedures. Cypress uses *AlpineFile.ReadPages* and *WritePages* to transfer a single page per call. When Cypress is called from Walnut, the message filing system, the average time for an

AlpineFile.ReadPages call varies between 20 and 40 m. The average time for an *AlpineFile.WritePages* call is generally about 10 ms. These times are for a Walnut database of about 5 mbytes.

We were also interested in Alpine's performance when performing sequential reads and writes, since an IFS replacement will have to do many of these. We measured the elapsed time to copy large files between a Cedar workstation and Alpine, using the Copy procedure included in the Alpine commands package. This procedure performs a loop. Each time through the loop, it calls *AlpineFile.ReadPages* or *Writepages*, where it is appropriate, transferring 12 pages in each call. Copying a 5-mbyte file from the Alpine server to a Cedar workstation (also a Dorado) in this way takes about 89 s, including the time to create the file on the workstation. So the Copy procedure copies approximately 450 kbits/s.

Mitchell and Dion ran some simple experiments to give some indication of the relative performance of XDFS and the Cambridge File Server [19]. We ran the same experiments on Alpine. First we give some background on the experiments, then tabulate the results, and finally interpret these results.

The experiments involve making various calls from a client machine to a server, and measuring the elapsed time of the procedure call on the client machine. Hence, this time includes time spent by communications software on the client machine, time spent transmitting bits both ways over the network, and time spent doing work on the server. The client machine is of the same type as the server (both Dorados in the case of Alpine). As stated earlier, XDFS ran on an Alto. It used the same model of disk drive and the same experimental Ethernet as Alpine does.

In Experiments 3 and 4 it was not possible to perform precisely the same operations on both XDFS and Alpine, because of differences between the systems. Therefore, in Table I we mark the Alpine results with an asterisk (*). The differences will be explained when interpreting these results below. See Table I.

Experiment 1 measures the time for a null call to the server. The call takes no parameters, returns no results, and does no work. Birrell and Nelson measured the time of 1.1 msec. for the Cedar RPC facility that Alpine uses [3]. The transmission time on the Ethernet is 130 ms for Alpine, and roughly the same for XDFS.

If XDFS and its client were running on Dorados, this would reduce the processor time for a null call and return to $(38 - 0.13)/8 = 4.7$ msec, versus about 1.0 msec for the Cedar RPC facility used by Alpine. This reflects Birrell and Nelson's careful implementation of Cedar RPC.

Experiment 2 measures the time for the null transaction. For Alpine the null transaction requires two remote procedure calls: *AlpineTransaction.Create* with *createLocalWorker* = *TRUE*, and *AlpineTransaction.Finish* with *requestedOutcome* = *commit* and *continue* = *FALSE*. For XDFS the null transaction also requires two requests.

Because Alpine notices that the transaction is read-only, it does not wait for the commit record to reach the disk before returning from *AlpineTransaction.Finish*. The Alpine cost is still greater than the average rotational latency of the disk drive because *AlpineTransaction.Finish* waits for a log record produced by *AlpineTransaction.Create* to reach the disk. For most real transactions this log writing takes place asynchronously during the transaction; for very

Table I

Experiment	XDFS	Alpine
1. Null call and return	38 ms	1.1 ms
2. Create Transaction		
3. Commit Transaction	360 ms	10 ms
4. Random read (average of 100)	76 ms	25 ms*
5. Random write (average of 100)	142 ms	9 ms*
Create Transaction		
Write 256 kbytes to existing file		
Commit Transaction	49.5 s	4.8 s (512 bytes/call)
		2.6 s (2048 bytes/call)
		3.2 s (8192 bytes/call)

short read-only transactions this (unnecessary) log synchronization determines the performance. The XDFS cost is much higher because XDFS makes several disk transfers to allocate and write an intentions list and perform full two-phase commit.

Experiment 3 measures the time for 100 reads to randomly chosen pages in an existing 256-kbyte file, then divides by 100 for the average time per read. The XDFS experiment performs 256 byte reads. This is not possible with Alpine's interface; hence the Alpine experiment performs 512 byte reads. This difference is probably negligible.

It is clear that under the experimental conditions, both systems can generally satisfy a random read request using only one disk transfer. It is difficult to compare these results more precisely because the times include unknown proportions of seek time, rotational latency, and compute time; the proportion of seek time for the two systems may be quite different because of differences in the way the systems allocate disk sectors to file pages.

Experiment 4 measures the time for 100 writes to randomly chosen pages in an existing 256-kbyte file, then divides by 100 for the average time per write. It does not include the time to commit the transaction. The XDFS experiment writes 256 bytes per request. This is not possible with Alpine's interface; hence the Alpine experiment writes 512 bytes per call.

The difference here is not negligible: XDFS reads a page (to retrieve the unmodified bytes), then allocates and writes a page, while Alpine simply performs a log write. XDFS would have saved at most 38 ms if the size of a write request had matched its page size (the time for a read in Experiment 3, minus the communication overhead), so the correct comparison is Alpine's 9 ms versus at least 104 ms for XDFS. A random Alpine write costs less than a read because the write involves only log writing, which is both sequential and asynchronous; a random XDFS write costs more than a read because the write involves a page allocation.

Experiment 5 measures the time required to create a transaction, write 256 kilobytes sequentially into an existing 256-kbyte file, and commit the transaction. For Alpine, this is 3 + 256/kb remote procedure calls: the same two *Alpine-Transaction* calls as for the null transaction, with an *AlpineFile.Open*, and a

sequence of *AlpineFile.Write* calls (each call writing kb kilobytes) sandwiched in between. We performed this experiment on Alpine with kb = $\frac{1}{2}$, 2, and 8.

When Alpine writes 512 bytes per call, the ratio of Alpine performance to XDFS performance is roughly the same as in Experiment 4 (after allowing for the reads that XDFS does not perform in Experiment 5). The time with 2048 bytes per call is roughly half of the time with 512; given the small pages used in Alpine, it is good to send a run of pages in a single call. The time increase with 8192 bytes per call might be related to the fact that the disk rotates more than once in the time it takes to transmit 8 kilobytes over the experimental Ethernet.

XDFS writes two copies of intentions list and file map pages; that slows it down on Experiments 2, 4, and 5. Does the extra writing give XDFS more reliability than Alpine with one-copy logging? XDFS writes both copies of the replicated structures on the same disk volume, so they are not guaranteed to survive all single storage medium failures. The XDFS file map contains vital information; the Alpine file map, implemented in the Cedar nucleus file system, is just a hint that the Cedar nucleus file system scavenger can reconstruct from information in disk sector headers. This suggests that Alpine with one-copy logging is about as reliable as XDFS.

Performance was not the deciding factor in replacing XDFS with Alpine, but the figures above confirm what our users observed: that using Alpine does improve performance. The improvement is partly due to Alpine's larger virtual and real memory, partly from Alpine's different algorithms, and partly from Alpine's faster processor.

7.5 Future Work

We took a detour to get applications running. Alpine's goals can still be met. Based on experience, there are still some changes and improvements we would make to Alpine.

An IFS-like hierarchical directory system and a Pup-FTP server have been coded as Alpine clients, but not yet integrated with the Alpine server. Also, as mentioned earlier, Alpine's implementation does not yet support backup or multiple volumes. When these tasks are complete, Alpine should be a viable replacement for IFS.

Experience with Alpine applications to date has suggested the following three changes to Alpine. All three changes relate to transactions.

Alpine's clients have made heavy use of the feature *AlpineTransaction.Finish*, which allows them to commit updates but immediately creates a new transaction, maintaining the consistency of their application data structures by not releasing locks on Alpine. This seems to be a fine way to operate when the application works on a private database, but using this feature reveals a problem with Alpine. Recall that the act of extending a file sets a write lock on an owner database page (to change the space in use for the file owner). This lock is not released by calling *AlpineTransaction.Finish* with *continue* = *TRUE*. Therefore, when the user runs two "private database" applications, the second application to extend a file will run into a lock conflict! Since Alpine writes the owner database by calling *AlpineFile.WritePages*, the best solution may be a change in the *AlpineFile* interface that allows lock requests to be specified as short or long in duration.

Every Alpine transaction writes one log record when it is created, and this limits the duration of a transaction. When this part of the log file needs to be reused, the transaction must go away. Our users would like to be able to run long read-only transactions without this interference. Alpine should provide this.

Alpine gives the client no way to find out why a transaction has aborted. Our users have convinced us that this is wrong. Alpine should log information about the cause of each transaction abort, and provide procedures so that a client can read this information.

A probable change to the Cypress database management system also suggests a change to Alpine. Cypress can undoubtedly improve its performance by increasing the size of database pages from 512 bytes to some larger unit, such as 2048 bytes. Alpine should support a *block size* file property so that Cypress will not be penalized with unnecessary locking overhead when it makes this change. The block size would be the number of pages locked by each fine-grained lock on a file. The default block size of one would result in page locking, as is provided now.

Another change is suggested by the design of Alpine's backup system. To dump a volume without disrupting transactions in progress, the backup system must be able to read files without setting read locks on them. Alpine should provide a new lock mode for this type of access, so that the backup system can read files using the *AlpineFile* interface.

We expect that further changes will be suggested by the use of Alpine as an IFS replacement. The high water mark optimization was designed to eliminate the need for unlogged file updates, as in the Cambridge File Server's standard files [19]; it will be interesting to see how well this succeeds.

8. CEDAR'S INFLUENCE ON ALPINE

Alpine was implemented in the Cedar language, and calls the Cedar nucleus and Cedar RPC facilities. The four novel features of Cedar that helped most in implementing Alpine were garbage collection and finalization; lightweight processes; Cedar RPC and its stub generator; and interface and implementation modules.

8.1 Garbage Collection and Finalization

The Cedar language's most significant extensions of Mesa are garbage collected storage and the option of runtime, rather than compile-time or bind-time, type discrimination and checking. The Cedar nucleus includes runtime support for storage allocation, garbage collection, and type discrimination. 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 [23]. (This structure is not unique to Cedar; Lisp Machine Lisp [29] and Interlisp-D [28] are also 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 *AlpineFile* procedures *ReadPages* and *WritePages*. 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 Cedar 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 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 accomplish 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. Experience with Alpine and other Cedar applications led to a significant reengineering 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 [16]. 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 inexpensive. 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 Cedar RPC

Overall we feel that RPC and Lupine, the stub generator, were a boon to Alpine. In spite of its restrictions on argument types, Lupine did not inhibit the design of our file system interfaces. Lupine translates roughly 600 lines of Alpine public interfaces into roughly 7000 lines of bug-free Cedar code that performs well.

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. It is possible that we could have maintained the clean separation between communications and file system without Lupine, even without adopting an RPC-style interface, but the existence of Lupine eliminated the possibility of being incorrect.

RPC implementations compatible with Cedar's have been produced for C, Interlisp, Mesa, and Smalltalk programs, so calls can cross between many programming environments.

While implementing Alpine, we noticed two inconvenient aspects of RPC. 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, the Cedar language provides no convenient way to manage and refer to a dynamically varying set 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.

We would have liked to have a debugger that understood RPC. Because RPC is a stylized form of communication, the Cedar debugger could have been extended to trace a call stack across machine boundaries. In many situations the rest of the server could have continued delivering service while a process that raised an exception was being debugged remotely.

With such a small user community, we have never attempted to support two versions of the Alpine public interfaces at one time. Lupine and RPC do not completely solve the difficult problem of protocol evolution, but we think they will help. One can easily imagine extensions to Lupine that would help even more, by generating a server stub that imports a new interface version and exports the old version (as long as the old is sufficiently similar to a subset of the new).

It is worth reemphasizing 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 the originating machine. A great advantage of 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 Interface and Implementation Modules

We feel that the interface module (borrowed from Mesa) is the strongest feature of the Cedar language. Cedar interfaces prevent errors by providing type checking across implementation module boundaries. Modules help you organize thinking and work; the entire design process for the Alpine system was organized around the writing of interface modules, and the implementation was also coordinated at that level. If Cedar hadn't already had interfaces, Lupine would have made it necessary to invent them.

Interface modules encourage the sort of clean decomposition of a large system that is necessary if it is to evolve. 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 tested this hypothesis in the Alpine project by converting Alpine to use a new Cedar nucleus file system that bore no resemblance to the previous file system, based on Pilot. Essentially the entire work of the conversion was confined to the Buffer component, as expected.

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. But this feature can be difficult to use; often a programmer must define a new interface just to transmit a few parameters to the package, and must also keep track of several files generated by the compiler in the instantiation process. The Cedar system modeller [17] is designed to solve these file management problems in building large Cedar programs. Lacking the modeler, we used module instantiation for one large package that was instantiated only twice. For a smaller package that was used in more places within Alpine, we instantiated it by copying its source code in a stylized way. This is okay as long as the package is stable.

ACKNOWLEDGMENTS

Many people have 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

1. BIRRELL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M. D. Grapevine: An exercise in distributed computing. *Commun. ACM* 25, 4 (Apr. 1982), 260–274.
2. BIRRELL, A. D. Secure communication using remote procedure calls. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 1–14.
3. BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
4. BOGGS, D. R., SHOCH, J. F., TAFT, E. A., AND METCALFE, R. M. Pup: An internetwork architecture. *IEEE Trans. on Commun. Com-28*, 4 (Apr. 1980), 612–624.
5. BORR, A. Transaction monitoring in ENCOMPASS: Reliable distributed transaction processing. In *Proceedings of the Seventh International Conference on Very Large Databases*, (Cannes, France, Sept. 1981), VLDB Endowment, Saratoga, Calif. 155–165.
6. BROWN, M. R., CATTELL, R. G. G., AND SUZUKI, N. The Cedar DBMS: A preliminary report. In *Proceedings of the ACM SIGMOD 1981 International Conference on Management of Data* (Apr. 29–May 1, Ann Arbor, Mich.), ACM, New York, 1981, 205–211.
7. CATTELL, R. G. G. Design and implementation of a relationship-entity-datum data model. Xerox PARC Tech. Rep. CSL-83-4, May 1983.
8. CLARK, D. W., LAMPSON, B. W., MCDANIEL, G. A., ORNSTEIN, S. M., AND PIER, K. A. The Dorado: A high-performance personal computer. Xerox PARC Tech. Rep. CSL-81-1, Jan. 1981.
9. ELHARDT, K., AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 503–525.
10. GESCHKE, C. M., MORRIS, J. H., AND SATTERTHWAITE, E. H. Early experience with Mesa. *Commun. ACM 20*, 8 (Aug. 1977), 540–553.
11. GRAY, J. Notes on database operating systems. In *Operating Systems—An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (Eds.), Springer-Verlag, New York, 1978, 393–481.
12. GRAY, J. N., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND

TRAIGER, I. The recovery manager of the system R database manager. *ACM Comput. Surv., 13*, 2 (June 1981), 223–242.
13. IBM Corp. Customer information control system/virtual storage (CICS/VS) version 1 release 5 system/application design guide. IBM Order Number SC33-0068-2, May 1980, Chapter 7.3.
14. ISRAEL, J. E., MITCHELL, J. G., AND STURGIS, H. E. Separating data from function in a distributed file system. Xerox PARC Tech. Rep. CSL-78-5, Sep. 1978.
15. LAMPSON, B. W. Toward a concise description of a large system. Xerox PARC Tech. Rep. CSL-83-15 (to be published).
16. LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (Feb. 1980), 105–117.
17. LAMPSON, B. W., AND SCHMIDT, E. E. Organizing software in a distributed environment. In *Proceedings of the ACM SIGPLAN 83 Symposium on Programming Language Issues in Software Systems* (San Francisco, June 1983), ACM, New York 1–13. Published as *SIGPLAN Notices 18*, 6 (June 1983).
18. LINDSAY, B. G., SELINGER, P. G., GALTIERI, C., GRAY, J. N., LORIE, R. A., PRICE, T. G., PUTZOLU, F., TRAIGER, I. L., AND WADE, B. W. Notes on distributed databases, IBM Res. Rep. RJ2571, 1979.
19. MITCHELL, J., AND DION, J. A comparison of two network-based file servers. *Commun. ACM 25*, 4 (Apr. 1982), 233–245.
20. MITCHELL, J. G., MAYBURY, W., AND SWEET, R. Mesa language manual, version 5.0. Xerox PARC Tech. Rep. CSL-79-3, Apr. 1979.
21. PETERSON, R. J., AND STRICKLAND, J. P. Log Write-Ahead Protocols and IMS/VS Logging. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (Atlanta, Ga.), ACM, New York, 1983, 216–243.
22. REDELL D. D., DALAL, Y. K., HORSLEY, T. R., LAUER, H. C., LYNCH, W. C., MCJONES, P. R., MURRAY, H. G., AND PURCELL, S. C. Pilot: An operating system for a personal computer. *Commun. ACM 23*, 2 (Feb. 1980), 81–92.
23. ROVNER, P. On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language. Xerox PARC technical report CSL-84-7 (to appear).
24. SCHROEDER, M. D., GIFFORD, D. K., AND NEEDHAM, R. M. A caching file system for a programmer's workstation. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* (to be published).
25. SWINEHART, D. C., ZELLWEGER, P. Z., AND HAGMANN, R. B. The structure of Cedar. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments* (Seattle, Wash., June 25–28), ACM, New York, 1985, 230–244.
26. SVOBODOVA, L. File Servers for Network-Based Distributed Systems. *ACM Comput. Surv.* (to be published). This is an updated version of IBM Res. Rept. RZ1258, Oct. 1983.
27. TEITELMAN, W. A tour through Cedar. *IEEE Software 1*, 2 (Apr. 1984), 44–73.
28. TEITELMAN, W., AND MASINTER, L. The Interlisp Programming Environment. *IEEE Computer 14*, 4 (Apr. 1981), 25–34.
29. WEINREB, D., AND MOON, D. Lisp machine manual, *MIT Artificial Intelligence Laboratory*, Nov. 1978.

Received September 1984; revised July 1985; accepted July 1985