CSL Notebook Entry

To:	CSL	Date:	September 17, 1981
From:	Alpine designers (M. Brown, K. Kolling, E.Taft)	Location:	PARC/CSL
Subject:	Alpine file server overview	File:	[Ivy]<Alpine>Doc>AlpineOverview.bravo

XEROX

Attributes: informal, technical, Alpine, Database, Distributed computing, Filing, Remote procedure calls

Abstract: The Alpine project will implement a file server. This document describes the facilities that Alpine will provide and gives some information on how Alpine will be implemented.

1. Introduction

The Alpine project will implement a file server. This document describes the facilities that Alpine will provide and gives some information on how Alpine will be implemented.

This memo takes the CSL context for granted. The reader should know what an IFS is and what a Juniper transaction is. To learn about the latter, read the CSL report by Israel, Mitchell, and Sturgis, "Separating Data From Function in a Distributed File System," CSL-78-5, September 1978.

2. Why a new file server?

CSL's style of file server usage is changing. The type of use that is on the decline is exemplified by the Alto FTP program and Tajo's FileTool: direct user interaction with a file server to accomplish file transfers between a workstation and a server. An IFS is well suited to this type of use.

We can see two styles of file access that will likely predominate in the future: access through a univeral file system and access through database systems.

A *universal file system* would support a uniform address space for files, in which the unique identifier of a file is independent of where the file is stored. This means (or at least makes it likely) that any permanent file will be stored on a file server. To implement this scheme with acceptable performance, the universal file system will manage the local disk of a workstation as a cache of recently-used files. To increase a file's availability, the system may replicate it in several servers. This is especially attractive in the case of an immutable file (a file that is written once and never updated thereafter.) The directory system (that implements the mapping from file string name to file unique identifier) of a universal file system should be a distributed database, since it has no logical connection to any particular file server or workstation.

An IFS is not well suited to support a universal file system. One reason is that FTP connections are expensive (an IFS supports at most nine of them.) The file transfer rate from IFS could be much higher without saturating the disk or the ethernet (the Alto is slow and its memory is small.) IFS also does not support transactions, which are useful for consistent replication of files.

Database systems support shared, structured databases. A database system may run on a workstation or may run as a server; the server architecture has several advantages. One advantage is that a

server can protect records or even fields of records in a database, rather than protecting information at the granularity of files. Another advantage is that a server can perform concurrency control on the basis of logical database units, rather than files or pages, thereby increasing the allowable degree of concurrency. In either case, the database system performs both sequential access to large sections of a file and random access to pages (or small blocks of pages.) A database system tends to allocate files in large units (disk extents, which are groups of adjacent cylinders), and wants logically adjacent pages to be physically adjacent most of the time.

An IFS is not well suited to support database systems; this was never a goal of IFS. Since IFS is based on the Alto operating system, its facilities for random file access (through Leaf) are an add-on. IFS does not support transactions, so each database system using IFS must implement its own facilities for concurrency control and recovery.

In the future, many CSL application programs will deal primarily with database systems instead of file systems. But it will take time to develop database systems in CSL that are sufficiently fast and reliable for this heavy use. Even in a database-oriented CSL, files are a convenient means of communication with the outside world. Hence a new file server should support the uses of files that are common today (source and object code, documents, images) as well as the needs of future database systems.

There are several reasons why it makes sense to build a new file server rather than converting Juniper to run in the Cedar/Pilot world. Converting Juniper would not be easy; a new system can be structured from the start to take advantage of virtual memory and a standard storage allocator. Juniper was not designed to support fast sequential transfers, which we now feel are important even in a database environment. Now that CSL has some experience with databases, we can design a file server that supports databases more effectively than Juniper does, while eliminating some Juniper features that database systems are not likely to need.

3. Alpine's scope

Alpine's scope is determined by the projected needs of CSL, as described above, and the interests and energies of the implementors. Here are our conclusions.

Included

Transactions. Alpine must implement atomic transactions, to support file replication and database systems. Transactions must be able to span multiple machines.

Access control. Alpine must implement some form of control over access to files. These access control facilities should be simple; there is little motivation for elaborate access control at the server level if the universal file system and database systems will be implementing their own access control policies.

Disk space accounting. Alpine must implement some form of control over the allocation of disk space to various users and projects. In the long term we expect that the main client of the space accounting facilities will be higher level facilities rather than individual users, since users won't want to be bothered with knowing where their files are actually stored.

Logical locking and recovery for database systems. Alpine must support the many database systems that will be written over the next few years. The simpler systems will rely on the locking and recovery facilities that are supplied implicitly by Alpine transactions, but other systems will want to perform their own concurrency control to improve performance.

Capacity, speed, reliability, availability. Alpine must meet performance goals of various kinds. A server should be able to maintain many (say 40 or 50) connections at once; the cost of an inactive connection must be low. The speed of both sequential and random (single page or short block of

pages) transfers should be good, making effective use of a large processor memory for buffering; the special case of whole-file transfers will continue to be important for some time to come.

Alpine must support server configurations that survive any single failure of the disk storage medium without any long-term loss of information from committed transactions. This degree of redundancy should not be a requirement for all servers, however.

Since storage medium failure is rare, recovery from it can be relatively slow (a few hours); recovery from crashes caused by software bugs should be much faster (5-10 minutes.) It should be possible to operate a server in a degraded mode when portions of it fail (say one volume or drive); it should be possible to move a logical disk volume from one server to another, and to store a volume offline.

Workstation file system. The Alpine project is constructing a file server, but there is no intrinsic reason why the system that is produced cannot run on a workstation. (The workstation must still contain a standard Pilot volume for code swapping, virtual memory, etc.) Allowing a workstation to contain an Alpine file system would support local databases. The workstation might also attempt to provide Alpine file services to the network, allowing a small system configuration that does not contain a dedicated file server. A workstation-based file system may be limited in some ways, for instance by the lack of independent disk volumes to improve performance, reliability, and availability, and the lack of an operations staff to perform backup tasks. When the requirements of a shared file server conflict with the requirements of a workstation file system, Alpine's design favors the shared file server.

Deferred

Archiving. A system for archiving files from the primary disk storage of a file server and for bringing archived files back again seems very valuable, and we should plan for it. But implementing an archive system is a major task and IFS has been successful without doing it, so Alpine should defer it until higher-priority goals have been reached. We do feel that the Alpine file organization (low-level naming by unique IDs, with high-level naming by a separate location-independent directory system) eliminates many of the problems that would make it difficult to implement a satisfactory archive system for IFS.

The requirements of archiving should have an impact on the design of our database systems. If optical disks become a reality it may become possible to hold the entire "archive" online, which will change the nature of archive systems. If a means of file replication is provided on top of Alpine, this might also serve as the archiving mechanism.

Excluded

Location-transparent file access. Alpine will not implement services such as volume or file location. A client of the Alpine interface will be aware at all times that he is communicating with a single server. We expect a more civilized interface to be built on top of this.

Directory system. Alpine will not implement a directory system. As we argued previously, directories should not be localized to particular servers but should instead be replicated and distributed across servers and workstations.

FTP access. The FTP protocol requires a directory on the file server.

It is possible that a single-server directory system and an FTP server might need to be built as part of the transition to Alpine. This will depend upon how fast other projects, particularly a universal file system, are able to progress. Some clients, such as the Cedar database management system, will be able to use Alpine immediately, without a directory system or FTP.

Continuous availability. Our environment does not demand continuous availability of a file server; we can tolerate scheduled downtime during off hours, and small amounts of downtime due to

crashes during working hours. If better availability has significant cost, we don't need it.

Guaranteed real-time response. Alpine's emphasis is on reliability and good average-case performance, and not on the guaranteed real-time response that an audio file server must provide.

4. Alpine's implementation strategies

Alpine uses Pilot to handle its disks. This means that a disk volume used by Alpine has Pilot disk format, as well as a higher-level structure defined by Alpine; Pilot disk utilities, such as the formatter and scavenger, work on Alpine volumes. Alpine does not use Pilot's implementation of transactions.

Alpine implements transactions using an online disk log. Most operations, such as writing a file page, are recorded in the log but not executed until after a transaction commits. Other operations, such as creating a file, are recorded in the log and executed immediately, and later undone if the transaction aborts. By writing the log to two independent log volumes, we can ensure that the effects of a transaction survive any single failure of storage medium while the transaction is being carried out; the log is then saved to tape for use in backup. The time required to restore lost information from log tapes is bounded by periodically copying entire volumes to backup packs; all of a server's volumes need not be copied at one time.

Using the log and deferring updates until commit is not free (though having lots of primary memory makes it more so), and we expect that some clients will not require it. Alpine offers a mode of file access in which updates are not protected by the log. A possible client of this mode is the file replication machinery, which might choose to log updates to the primary copy of a file only, and use this copy to fix the others if they fail. In case the server crashes while such an unprotected update is in progress, the file being updated is marked bad. Alpine also optimizes a case (updating a file past the highest page written in any earlier transaction) that includes FTP style transfers. This optimization makes logging and file update occur in parallel, instead of deferring the file update until after commit.

Alpine clients are identified by RName and authenticated through Grapevine. Alpine uses Grapevine to help implement file access control.

The core of Alpine is a file system that exports a simple interface, usable by clients on its own machine. One such client is a module generated by the Cedar remote procedure call (RPC) facility. To access a remote Alpine server, a client on a remote machine must load a stub implementation of the Alpine file system interface; the RPC facility also generates this module. RPC will support authenticated, encrypted conversations over the Ethernet, with especially good performance for calls on the local net. The Cedar RPC facility was designed to support Alpine as one of its first clients.

5. Alpine objects

In this section we shall informally describe the primary objects that are visible to a client of Alpine. These are the persistent objects *server*, *log*, *volume*, *file*, *owner*, *volume group*, and *transaction*, and the volatile object *open file*. In places where it clarifies things we shall mention some aspects of the implementation.

A *client* of an Alpine server, meaning a program that calls through Alpine's interface, is identified by the RName of an individual.

An Alpine *server* consists of a single log L and a set V of (logical disk) volumes. All recovery information that is recorded for volumes in V is written to L. A server is identified by the unique ID of its log L.

We generally expect a machine running Alpine to contain a single Alpine server. Two servers on the same machine communicate as if they were remote from each other (e.g. they follow the distributed two-phase commit protocol if a transaction whose coordinator is on one server makes updates on both servers.) One can imagine situations in which one log is a bottleneck and configuring multiple servers on one machine is the best way to solve the problem, but these situations are likely to be rare in our environment.

A *log* is a Pilot logical volume containing a single large file that is managed as a ring buffer of pages. The unique ID of a log is the unique ID of the Pilot logical volume containing it. Pages of log are always written in ascending order according to page number, until the log fills and writing starts again from the beginning. Information in a log is used for online recovery from server crashes and for undoing the effects of aborted transactions. In server configurations that include backup facilities, each log page is recorded on tape before the disk copy is overwritten; this tape is called the *archive log*. (The archive log actually includes only the log records that are relevant to media recovery.)

A *volume* is a Pilot logical volume with some added higher-level structure. The unique ID of a volume is the unique ID of the Pilot logical volume. We expect most volumes to be entire disk packs.

For an Alpine server to tolerate all possible single failures of storage media, it must write two independent copies of the log (consider a failure of the log between the time of commit and the time that the commited transaction's writes are propogated to a file.) These log copies must be stored on different packs, but each copy may stored on the same pack as some logical volume. Logging is most efficient if two drives are dedicated to the log copies, since in this case the speed of logging is not limited by disk arm motion. (Continuous logging at high rates actually requires three drives, so that one drive can be dumping the log to tape as the other two accumulate it. We do not plan to support this feature.)

A volume can be moved from one server to another. This operation requires a "volume quiesce", which means aborting all transactions that request access to the volume, performing all committed updates to the volume, and taking the volume offline. (A quiesced volume may be stored offline.) In principle, a volume quiesce is not required if a copy of the log is stored on the same pack as the volume, but it is not clear that the option of moving a volume without a quiesce is worth supporting.

A *file* is a set of property values and a sequence of 512 byte pages, indexed from zero. A file is implemented from a Pilot file (some number of "leader pages", invisible to the client, may be used to store property values) and a file's unique ID is the Pilot file's ID. A file is contained in a volume.

The set of file properties is fixed (not expandable.) The set includes the Pilot file attributes *type* and *immutable*, and other properties such as a string name, a byte length, a create time, and so on.

One of a file's properties is its *owner*. An owner is an entity indentified by an RName, such as "McCreight.pa" or "CedarImplementors^.pa". The disk space occupied by a file is charged against its owner's space quota.

Two other file properties are its *read list* and its *modify list*. Each of these is a list of RNames, such as (CSL^.pa Wick.pa). For implementation simplicity these lists are limited to contain at most two RNames, except that the owner of a file, and the special RName "*" meaning "world", do not count against the two RName limit. An Alpine client may read a file if he is contained in (the closure of) one element of its read list. An Alpine client may read or modify a file if he is contained in (the closure of) one element of its modify list.

A server often contains several volumes that are equivalent from the point of view of its users. In particular, a user may not care which volume he creates a file on. For this reason, each server's volumes are partitioned into one or more *volume groups*. A group contains one or more volumes, and every volume belongs to exactly one group. Disk space quotas for owners are maintained for

volume groups rather than individual volumes. A create file call may specify a volume group, instead of a specific volume; the server decides which volume to create the file on, and informs the client. We expect an entire volume group to go online or offline together. In some cases it will be possible to operate with some volumes of a group offline.

A *transaction* is identified by the ID of a server (log) and a unique sequence number on that server. The server named in the transaction ID is the coordinator of the transaction. In principle, this server can always respond to queries of the form "did transaction t commit or abort?" In practice, the server will respond "don't know" if the transaction is very old (no record of it is online.) We are interested in the possibility of having the coordinator store replicated commit records on other servers, but do not plan to implement this feature right away.

An *open file* is essentially an association between a file, a client, and a transaction. All calls on an Alpine server that access a file require that the file be opened. Access control and file-level locking is performed at file open time.

Important omissions

We have not described the Alpine objects that an ambitious database system might use in addition to files: locks and log records. This is because to do so would greatly enlarge this document. [Ivy]<Alpine>Doc>LockConcepts*.bravo (* = 0, 1, ...) describes locks, and [Ivy]<Alpine>Doc> DBMSRecovery.bravo describes logging (but less definitively.)

6. Where to learn more about Alpine

The memos [Ivy]<Alpine>Doc>FileStore*.bravo (* = 0, 1, ...) describe successive iterations of the public interfaces to Alpine. Read the most recent version of this document to learn the specifics of the interfaces as they exist today. The public Cedar interfaces themselves (AlpineFile, AlpineAccess, AlpineTransaction) are stored on [Ivy]<Alpine>Defs>. A description of the interfaces from the implementation's point of view is given on [Ivy]<Alpine>Doc>FileStoreInternal*.bravo.