Inter-Office Memorandum

To	Alpine project	Date	October 13, 1981
From	Mark Brown and Ed Taft	Location	PARC/CSL
Subject	FileStore public interfaces, version 8	File	[Ivy]<Alpine>Doc>FileStore8.press (from FileStore8a.bravo, b.bravo)

XEROX

Attributes: informal, draft, technical, Alpine, data base, filing, remote procedure calls

Abstract: The FileStore interfaces are the lowest-level public interfaces to the Alpine file system. This memo is intended to evolve into the client programmer's documentation for the Mesa definitions files AlpineEnvironment, AlpineAccess, AlpineFile, and AlpineTransaction.

See the end of this memo for a change history and a list of unfinished business. <We use angle brackets to enclose comments, like this one, that should disappear once uncertainty in the design is resolved.>

1. Introduction

This memo documents the lowest-level file system interfaces seen by Alpine clients; these interfaces are referred to collectively as FileStore. The FileStore interfaces are intended to support higher-level facilities such as the Cedar data base and the universal file system, and not to support application-level clients directly. Multiple instances of FileStore may be imported by a single client, some representing local file systems and others representing remote servers that are accessed through remote procedure calls (RPC).

We also use the term FileStore to mean a server system exporting the FileStore interfaces. A FileStore consists of a single log, used to implement atomic transactions on files, and a set of logical disk volumes containing files. A log is stored on disk for online recovery from client-aborted transactions and soft failures, and is transferred to offline storage for use by the backup system in case of hard media failures. A volume may be quiesced and moved from one FileStore to another, or stored offline for extended periods. We generally expect a server machine to contain a single FileStore. A single server machine containing more than one FileStore is advantageous if the server is a high-speed processor with many disk arms, in order to allow multiple logs per processor and thereby increase the maximum transaction rate.

FileStore includes the procedures used for inter-server communication, as required for the coordination of transactions involving multiple servers. The transaction coordination primitives take advantage of important special cases such as single file system transactions and read-only transactions. Some aspects of the interface are designed with an eye to supporting replicated commit records as a future option, but the interface does not fully support them now.

The FileStore interfaces are the only ones seen by clients of an Alpine server. This includes clients that may share the server machine, such as a directory or data base system.

2. Organization

There are three main FileStore interfaces: AlpineAccess, AlpineFile, and AlpineTransaction. Additionally, there is a public definitions file, AlpineEnvironment, that declares public types, constants, and other things used in one or more of the interfaces. These definitions are also declared

(indirectly) in the interfaces that use them, so most clients should not need to refer to AlpineEnvironment directly; thus, for example, AlpineTransaction.TransID and AlpineFile.TransID are the same type, which is actually defined in AlpineEnvironment.

The AlpineAccess interface deals with all matters of access control, including initiation of RPC conversations, authentication, and control over access to files and other objects. Alpine depends very heavily on Grapevine in implementing these functions: clients are identified by Grapevine registered names (RNames), and access to files is controlled by treating Grapevine groups as access control lists. Additionally, each Alpine server maintains a structure of its own called the Owner data base, whose main purpose is to provide administrative controls such as disk space accounting.

AlpineFile provides all operations on files themselves, including file creation and deletion, reading and writing data, and manipulating any of a fixed set of file properties.

AlpineTransaction contains everything to do with transactions. An Alpine server may be asked to be the coordinator for a new transaction or a worker for an existing one; this interface includes operations for both purposes.

<The interfaces specified below restrict themselves to the procedure call semantics imposed by RPC. In particular, address-containing arguments and results are dereferenced by RPC (to a single level) and passed by value. Exported variables are not supported. REF ANY is not supported. Procedure arguments are not supported initially, though they may be eventually. At the moment, the RPC designers believe that SIGNALs and ERRORs will work across remote interfaces in the standard fashion, with the exception that signals not declared in the remote interface will not be accessible by name to the client. Therefore we have defined these interfaces to use signals in the conventional ways (for abstraction failures, calling errors, and information passing); and we intend never to allow non-interface signals to escape through the public interfaces. If the RPC designers decide not to support signals after all, we can fall back to a scheme of returning specific outcomes that the client must test on each call.>

3. RPC binding and authentication

Initiating a conversation with a remote Alpine server consists of two steps, *binding* and *authentication*. Both steps make extensive use of facilities provided by the CedarRPC interface. (CedarRPC is a Cedar veneer over RPC, as RPC is intended to be usable by non-Cedar clients as well.) Enough of RPC is shown below to give a general idea of the necessary operations; consult the current version of RPC.mesa and CedarRPC.mesa for more details.

3.1. Binding

A client binds to a local *stub implementation* of a remotely-exported interface using the normal Mesa facilities (binder, loader, etc.) For example, the stub implementation for AlpineAccess is called AlpineAccessRpcClientImpl, and is instantiated on the *client* machine.

The stub implementation does not correspond to any real implementation until the client invokes the appropriate RPC runtime machinery. Each stub implementation, in addition to exporting the intended interface, also exports an auxiliary interface used to invoke RPC runtime operations on behalf of that stub. For example, corresponding to the AlpineAccess interface there exists an AlpineAccessRpcControl interface.

```
CedarRPC.InterfaceName: TYPE = RECORD [
  type: Rope.Ref, -- e.g., "AlpineAccess"
  instance: Rope.Ref _ NIL, -- e.g., "MontBlanc.Alpine"
  version: VersionRange _ matchAllVersions];

CedarRPC.RPCZones: TYPE = RECORD [
  safeParameterStorage: ZONE _ NIL,
  heapParameterStorage: UNCOUNTED ZONE _ NIL,
```

```
  mdsParameterStorage: MDSZone _ NIL];

CedarRPC.Import: PROCEDURE [proc: ImportProc, interface: InterfaceName,
  zones: RPCZones _ [NIL, NIL, NIL]];

CedarRPC.ImportProc: PROCEDURE [ ... ];

AlpineAccessRpcControl.ImportInterface: ImportProc;
    --! RPC.ImportFailed[why: { ... }];
```

Each stub exports an ImportInterface procedure which, when called, causes the stub to be associated with a specific instance of a real implementation identified by the InterfaceName; by convention, the InterfaceName.instance specifies the Grapevine RName of the desired Alpine server. Thereafter, any call on a procedure in the main interface (e.g., AlpineAccess) will automatically turn into a remote procedure call to the actual implementation.

<The optional *zones* argument deals with some details of the RPC runtime machinery that haven't been specified yet.>

ImportInterface is an unsafe'' procedure; Alpine clients should ordinarily invoke it through the CedarRPC veneer. For example:

```
CedarRPC.Import[proc: AlpineAccess.ImportInterface, interface:
  ["AlpineAccess", "MontBlanc.Alpine"]];
```

Note that each remote interface must be imported in this way; so, for example, to import AlpineAccess, AlpineFile, and AlpineTransaction from a particular Alpine server requires invoking the ImportInterface procedure in each of the three corresponding stubs.

ImportInterface is strictly a runtime binding operation, and does not give rise to any communication between the client and the remote implementation. There *is* communication between the local and remote RPCs, but it is invisible to the client and implementor of the interface being bound. In particular, the client and the implementor are not authenticated to each other (except for RPC's guarantee that they are type-compatible). Authentication is the responsibility of the client and implementor, though RPC does provide some assistance as described in the next section.

A runtime binding may be broken by calling, for example:

```
AlpineAccessRpcControl.UnimportInterface: PROCEDURE;
```

This causes the local and remote RPCs to forget about the association between the client and implementation. A client that knows it is finished dealing with a particular instance of the interface should invoke this to conserve resources in the RPC machinery.

RPC signals relevant to Alpine clients are:

```
RPC.ImportFailed: ERROR [why: ImportFailure];
RPC.ImportFailure: TYPE = { ... };

RPC.CallFailed: ERROR [why: CallFailure];
RPC.CallFailure: TYPE = { ... };
```

CallFailed may be raised by *any* remote call; it indicates a breakdown in communication or in the binding with the remote implementor. This will not be mentioned in the interface descriptions in the remainder of this document.

3.2. Conversation initiation and authentication

Before an Alpine server will grant service to a client, the client must be authenticated. <This is based on the protocol described in [Needham & Schroeder, 1978]; it makes use of some Grapevine facilities that don't exist yet, so it is somewhat speculative. The initial implementation is likely to be degenerate.>

The client program (or user on whose behalf it is working) is identified by a Grapevine RName, and has a secret key known only to it and to the Grapevine authentication servers. RPC provides a facility that takes this information, authenticates the client and server to each other, and establishes secure communication using a temporary *conversation key* known only to client and server.

```
CedarRPC.Principal: TYPE = Rope.Ref;

CedarRPC.EncryptionKey: TYPE [4];

CedarRPC.ConversationID: TYPE [2];

CedarRPC.StartConversation: PROCEDURE [caller: Principal, callerKey:
  EncryptionKey, callee: Principal] RETURNS [conversation:
  ConversationID];
    --! <lots of things can go wrong, but none are specified yet>;
```

The *caller* and *callee* are the RNames of the client and server respectively, and *callerKey* is *caller*'s encryption key. An EncryptionKey may be derived from a text password by calling:

```
CedarRPC.MakeKey: PROCEDURE [text: Rope.Ref] RETURNS [EncryptionKey];
```

StartConversation performs the following operations. First, it communicates with a Grapevine authentication server to obtain a conversation key and an *authenticator* consisting of the conversation key and client's RName encrypted by the server's key. It then allocates a ConversationID (whose purpose will be explained shortly), and registers the ConversationID and conversation key with the local RPC runtime machinery. Finally, it communicates the ConversationID and authenticator to the server's RPC runtime machinery. This is all the information required to carry on a subsequent secure conversation.

That conversation is facilitated by the following RPC convention: if the first argument of a remote procedure call is of type ConversationID, then RPC will ensure that the communication between client and server machines (for that call) is encrypted by the key corresponding to that ConversationID.

Thus, all procedures exported by Alpine FileStore interfaces require a ConversationID as their first argument. This ConversationID serves two purposes: it causes RPC to encrypt the communication, as just explained; and on the server side it identifies the client of each call.

A few remarks may be made about this organization. A ConversationID represents nothing more than an authenticated communication path between a pair of principals; there is no strong association between conversations and anything else (interfaces, processes, etc.) So a single ConversationID can be (and typically will be) used to make calls to multiple remote interfaces, such as the AlpineAccess, AlpineFile, and AlpineTransaction interfaces exported by one Alpine server. A single remote implementation can support conversations with multiple clients, each identified by its own ConversationID. Multiple client programs can communicate with a given Alpine server using the same or different ConversationIDs, depending on whether they are mutually cooperative or suspicious.

```
CedarRPC.EndConversation: PROCEDURE [conversation: ConversationID];
```

EndConversation cause the remote and local RPCs to forget about a conversation (i.e., about an association between ConversationID and conversation key). A client that knows it is finished with a conversation should invoke this to conserve resources in the RPC machinery.

4. AlpineAccess interface

Mesa definitions in this section are declared in AlpineAccess except where explicitly qualified.

4.1. Access control

Access to files is controlled by access control lists, which contain Grapevine RNames that may be groups. Client membership in access control lists is determined by consulting Grapevine; the Grapevine facilities are not described here.

```
AccessList: TYPE = LIST OF RName;

RName: TYPE = GVName.RName; -- = Rope.Ref
```

Each file has two access control lists that control reading and modifying that file. These access control lists are file properties that may be manipulated via the AlpineFile interface, described in a later section.

Each file also has an *owner* property, which is a single name (usually but not necessarily an RName; see below). A client whose RName matches the file's owner property (or is a member of the owner's create list see below) is permitted to change the file's access control lists.

Alpine's initial implementation permits an access control list to consist of at most two RNames. However, if a file's owner is a member of an access control list for that file, it does not count against the two RName limit. Additionally, there is a pseudo-RName "World" (abbreviated "*") that includes any authenticated client as a member; and there is another pseudo-RName "Owner" that is equivalent to the name of the file's owner. These pseudo-RNames do not count against the two RName limit.

Each FileStore has a special "Alpine wheels" access control list whose RName is part of the FileStore's global state. Membership in this group permits unlimited access to files and other objects, regardless of their access control lists; additionally, there are some administrative and debugging functions that can be invoked only by Alpine wheels.

To prevent accidental use of this dangerous capability, the client's membership in the Alpine wheels group is not noticed until explicitly enabled. Wheel membership is enabled or disabled by calling:

```
AssertAlpineWheel: PROCEDURE [conversation: ConversationID, enable:
  BOOLEAN];
    --! AccessFailed {alpineWheel};
```

4.2. Owner data base

Consumption of disk space in an Alpine server is controlled on the basis of information in an *owner data base*. An owner is simply a name (usually but not necessarily an RName) associated with a fixed set of information: a disk space quota, present consumption, and two access control list specifying who may create files for that owner (i.e., allocate disk space against the owner's quota) and who may change the owner information itself.

```
String: TYPE = Rope.Ref;

OwnerName: TYPE = String;
```

For the purposes of disk space accounting, several volumes on a given server may be grouped into a single *volume group*, which has associated with it a single owner data base. Permission to allocate space on behalf of an owner applies to any volume in the group. Additionally, when creating a file, a client may identify a volume group rather than an individual volume, leaving the choice of volume up to the server; this is discussed in the AlpineFile section.

Operations on the owner data base are included in the AlpineAccess interface. However, they are seldom of interest to most clients (and most of them require system administrator privileges); so their description is deferred to a later section.

5. AlpineFile interface

Mesa definitions in this section are declared in AlpineFile except where explicitly qualified.

5.1. File system organization

The protocol described in section 3 binds a client to a specific FileStore implementation, identified by an RName. Each FileStore consists of a *log*, used to implement atomic transactions, and some number of *volumes*, possibly organized into *volume groups*. A volume is analogous to a Pilot logical volume'': in its quiescent state it is a self-contained file system. The set of volumes composing a FileStore may change dynamically as volumes are mounted and dismounted, though the rate of change is expected to be slow.

```
FileStore: TYPE = RName;

VolumeID: TYPE = RECORD [VolOrVolGroupID];

VolumeGroupID: TYPE = RECORD [VolOrVolGroupID];

VolOrVolGroupID: TYPE [5];
```

A *file* is a sequence of fixed-size pages stored on a single volume. A file is identified by a unique *FileID*, and its pages are numbered consecutively from zero.

```
FileID: TYPE [5];

PageNumber: TYPE = [0..LAST[LONG INTEGER]];

PageCount: TYPE = LONG INTEGER;
```

5.2. Open files

A client desiring to gain access to a file must first open it. Opening a file serves two purposes. First, it defines a point at which file access control and whole-file locking are done. Second, it associates a brief handle, the OpenFileHandle, with a considerable amount of state that is thereafter kept by the server and need not be supplied in each call. That is, an OpenFileHandle represents a single client's access to a single file under a single transaction.

```
ConversationID: TYPE = RPC.ConversationID;

OpenFileHandle: TYPE [2];

TransID: TYPE [9];

AccessRights: TYPE = {readOnly, readWrite};

LockMode: TYPE = {read, update, write, intendRead, intendUpdate,
  intendWrite, readIntendUpdate, readIntendWrite};

LockOption: TYPE = RECORD [
  mode: LockMode,
  ifConflict: {wait, fail} _ wait];
```

```
RecoveryOption: {log, noLog};

Open: PROCEDURE [conversation: ConversationID, trans: TransID, volume:
  VolumeID, file: FileID, access: AccessRights _ readOnly, lock:
  LockOption _ [read, wait], recoveryOption: RecoveryOption _ log]
  RETURNS [OpenFileHandle];
    --! AccessFailed {fileRead, fileModify}, LockFailed, OperationFailed
      {fileImmutable}, PossiblyDamaged, Unknown {volumeID, fileID,
      transID};
```

Opens the existing file described by *volume* and *file* for access under transaction *trans*. (Operations dealing with transactions are in the AlpineTransaction interface, described in section 6.)

Pilot does not presently have any notion of volume-relative FileIDs; the *volume* argument is intended for future use in selecting among instances of a replicated file. In anticipation of this, Alpine now requires that the client present the correct *volume* for *file*, and raises Unknown[fileID] if it is incorrect.

The client is required to be a member of the access control list implied by *access* (read or modify); and if *access*=readOnly, the client is restricted to read-only operations on that OpenFileHandle.

The entire file is locked in *lock*.mode. If the file cannot be locked in that mode, Open either waits until the lock can be set or raises LockFailed[conflict] immediately, depending on *lock*.ifConflict. (The semantics of LockModes are discussed in section 6.2.)

Ordinarily, modifications to the file are protected against transaction aborts, crashes, and media failure by the log mechanism; logging may be disabled by passing *recoveryOption*=noLog. If a file was previously updated with *recoveryOption*=noLog and that transaction failed in the middle, Open will raise the signal PossiblyDamaged; this signal may be RESUMEd by the client. This feature exists primarily for the benefit of a (future) file replication facility: file replication does not require the protection provided by the log mechanism, and should not incur its cost. <There may be more restrictions on use of noLog; it may require an enabling file property.>

5.3. File creation

```
standardFile: File.Type = ??;

Create: PROCEDURE [conversation: ConversationID, trans: TransID, volume:
  VolOrVolGroupID, owner: OwnerName, initialSize: PageCount, type:
  File.Type _ standardFile, recoveryOptions: RecoveryOption _ log]
  RETURNS [OpenFileHandle];
    --! AccessFailed {ownerCreate, spaceQuota}, Unknown {volumeID,
      transID, owner};
```

This procedure creates and opens a new file under transaction *trans*. The *volume* argument may be either a specific Volume or a VolumeGroup; in the latter case, the server will choose among the volumes of the group.

The initial size of the file is specified by *initialSize*; clients are encouraged to create files of the required size rather than growing them piecemeal. The space is charged against *owner* in the owner data base; the client is required to be a member of *owner*'s create access control list, and the allocation must not exceed *owner*'s quota. Note that allocation is consumed at the moment a file is created; but when a file is deleted, the allocation is not credited with the freed pages until transaction commit time.

The file is created with a Pilot FileType of *type* <presumably restricted by Alpine>, and with Pilot attributes *immutable*=FALSE and *temporary*=FALSE. Since the FileStore is transaction-based, the Pilot *temporary* attribute is not of any use. All file properties are set to default values <to be determined>; they may be changed by means of the procedures described below.

If the call is successful, it returns an OpenFileHandle with *accessRights*=readWrite and *lockMode*=write.

5.4. Operations on OpenFileHandles

The following procedures access the volatile state associated with an OpenFileHandle:

```
GetVolumeID: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle] RETURNS [volume: VolumeID];
    --! Unknown {openFileHandle};

GetFileID: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle] RETURNS [file: FileID];
    --! Unknown {openFileHandle};

GetTransID: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle] RETURNS [trans: TransID];
    --! Unknown {openFileHandle};

GetAccessRights: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle] RETURNS [access: AccessRights];
    --! Unknown {openFileHandle};

GetRecoveryOption: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle] RETURNS [recoveryOption: RecoveryOption];
    --! Unknown {openFileHandle};

GetLockOption: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle] RETURNS [lock: LockOption];
    --! Unknown {openFileHandle};

SetLockOption: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, lock: LockOption];
    --! LockFailed, Unknown {openFileHandle};
```

SetLockOption actually changes the file lock, so it may wait or fail according to *lockOption*.ifConflict. File locks may only be upgraded by this means; attempts to downgrade a lock are ignored.

```
Close: PROCEDURE [conversation: ConversationID, handle: OpenFileHandle];
    --! Unknown {openFileHandle};
```

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

Files are automatically closed by AlpineTransaction.Finish called with *requestedOutcome*=abort or commitAndTerminate, but are left open by *requestedOutcome*=commitAndContinue.

5.5. Operations on open files

```
MakeImmutable: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle];
    --! AccessFailed {handleReadWrite}, LockFailed, OperationFailed
      {fileImmutable}, Unknown {openFileHandle};
```

First locks the entire file in write mode (if it isn't already opened that way); then sets the file's *immutable* attribute to TRUE. An immutable file is permanently read-only, and the OpenFileHandle is immediately modified to have *access*=readOnly to reflect this fact.

```
Delete: PROCEDURE [conversation: ConversationID, handle: OpenFileHandle];
    --! AccessFailed {fileModify}, LockFailed, Unknown {openFileHandle};
```

First locks the entire file in write mode (if it isn't already opened that way); then deletes the file. The client is required to be on the file's modify access control list; however, it is *not* necessary that the OpenFileHandle have *access*=readWrite (otherwise there would be no way to delete an immutable file). The OpenFileHandle is made invalid by this procedure.

```
PageRun: TYPE = RECORD [
  firstPage: PageNumber,
  count: CARDINAL _ 1];

VALUEPageBuffer: TYPE = DESCRIPTOR FOR ARRAY OF UNSPECIFIED;

RESULTPageBuffer: TYPE = DESCRIPTOR FOR ARRAY OF UNSPECIFIED;

ReadPages: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, pageRun: PageRun, pageBuffer: RESULTPageBuffer, lock:
  LockOption _ [read, wait]];
    --! LockFailed, OperationFailed {nonexistentFilePage}, Unknown
      {openFileHandle};
```

Reads data from the pages described by *pageRun* from the file associated with *handle*, and puts it contiguously into (client) memory starting at *pageBuffer*. The *lockOption* argument specifies the page-level locking to be performed; for a discussion of this and its relationship to file-level locking, see section 6.2. Note: the VALUE and RESULT prefixes conform to an RPC convention for dereferencing of pointer arguments: a VALUE is dereferenced at call time, whereas a RESULT denotes a place where a result is to be put at return time.

```
WritePages: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, pageRun: PageRun, pageBuffer: VALUEPageBuffer, lock:
  LockOption _ [update, wait]];
    --! AccessFailed {handleReadWrite}, LockFailed, OperationFailed
      {nonexistentFilePage}, Unknown {openFileHandle};
```

Writes data from (client) memory starting at *pageBuffer* onto the pages described by *pageRun* of the file associated with *handle*.

```
LockPages: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, pageRun: PageRun, lock: LockOption _ [update, wait]];
    --! LockFailed, Unknown {openFileHandle};

UnlockPages: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, pageRun: PageRun];
    --! Unknown {openFileHandle};
```

Pages are ordinarily locked automatically as a side-effect of calling ReadPages and WritePages, and unlocked by terminating the transaction (see section 6.2 for details). However, it is occasionally useful to lock pages in advance of doing I/O to them; this may be accomplished by calling LockPages. UnlockPages may be called to remove a *read* lock prior to the end of the transaction; it is the client's responsibility to ensure consistency by avoiding any subsequent updates that depend on the data formerly protected by the read lock. Update and write locks cannot be removed by UnlockPages; attempts to do so are ignored.

```
GetSize: PROCEDURE [conversation: ConversationID, handle: OpenFileHandle,
  lock: LockOption _ [read, wait]] RETURNS [size: PageCount];
    --! LockFailed, Unknown {openFileHandle};
```

```
SetSize: PROCEDURE [conversation: ConversationID, handle: OpenFileHandle,
  size: PageCount, lock: LockOption _ [update, wait]];
    --! AccessFailed {handleReadWrite, spaceQuota}, LockFailed, Unknown
      {openFileHandle};
```

Obtains or changes the size of the file. The file's size is locked by *lock*; additionally, decreasing a file's size locks the entire file in the same mode. SetSize appropriately adjusts the disk space charged to the file's owner; the *handle* must have *access*=readWrite; however, it is *not* required that the client be a member of the owner's create access control list. Note that allocation is consumed immediately when a file's size is increased; but when the size is decreased, the allocation is not credited with the freed pages until transaction commit time.

```
GetHighWaterMark: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, lock: LockOption _ [read, wait]] RETURNS
  [highWaterMark: PageCount];
    --! LockFailed, Unknown {openFileHandle};

SetHighWaterMark: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, highWaterMark: PageCount, lock: LockOption _ [update,
  wait]];
    --! AccessFailed {handleReadWrite}, LockFailed, Unknown
      {openFileHandle};
```

A file may contain pages whose contents are undefined. For example, when a file is created, all its pages have undefined contents; and when SetSize is used to increase the length of a file, the newly-allocated pages have undefined contents. The boundary between the defined and undefined portions of a file is called its *high water mark*.

Alpine maintains the high water mark so as to gain an important performance advantage: page writes beyond the high water mark are performed immediately rather than being deferred until the transaction is committed. This substantially reduces the number of disk transfers that the server must perform; it is intended to improve the performance of sequential bulk transfers.

The high water mark is advanced automatically by page writes beyond the previous high water mark; it may also be adjusted by the client by calling SetHighWaterMark. Decreasing the high water mark is a way of declaring that some portion (or all) of a file's contents are no longer interesting. Note that changing the high water mark, like all update operations, is deferred until the end of the transaction. So for a decrease in the high water mark to have the performance benefit described above, the transaction must first be committed.

5.6. File properties

Associated with each file is a fixed set of *file properties*. These include the underlying Pilot *type* and *immutable* (but not *temporary*) properties. Additionally there are several other properties that are expected to be generally useful; these include access control lists, byte length, string name, create time, and version.

File properties are read and written under a transaction, just like file data; properties may be individually locked or be locked implicitly by a file lock. In the initial implementation of Alpine, all properties with the exception of *version* are treated together with respect to locking. So, for example, writing any property will lock out reads of *any* property of the same file by another transaction.

Each file has an *owner*, which is a single OwnerName, and *read* and *modify* access control lists, each of which is a list of RNames. These were described in section 4.

The *byte length* of a file is a LONG INTEGER associated with the file. FileStore does not enforce any properties of a file's byte length; in particular, the byte length has nothing to do with the high water mark described in section 5.5.

The *string name* of a file consists of at most maxStringNameChars characters. FileStore does not enforce any properties of the string name.

The *create time* is a System.GreenwichMeanTime <or perhaps a full UniqueID?>, which by convention identifies the time at which the *information* in the file was created. FileStore does not enforce any properties of the create time. We discourage the use of create times to identify files; we provide versions (below) for that purpose.

The *version* of a file is a LONG INTEGER whose value is the number of committed transactions that have modified the file in any way (via WritePages, SetSize, SetByteLength, MakeImmutable, etc.) A single transaction may also increment a file's version by a specified amount. The version property is provided to allow remotely stored copies of non-immutable files to be validated; and multiple incrementing of the version property is provided to allow out-of-date replicas to be updated in a single transaction.

```
Property: TYPE = {type, immutable, version, owner, readAccess,
  modifyAccess, byteLength, stringName, createTime};

PropertyValuePair: TYPE = RECORD [
  SELECT property: Property FROM
    type => [type: File.Type],
    immutable => [immutable: BOOLEAN],
    version => [version: FileVersion],
    owner => [owner: OwnerName],
    readAccess => [readAccess: AccessList],
    modifyAccess => [modifyAccess: AccessList],
    byteLength => [byteLength: ByteCount],
    stringName => [stringName: String],
    createTime => [createTime: System.GreenwichMeanTime],
    ENDCASE];

ByteCount: TYPE = LONG INTEGER;

maxStringNameChars: CARDINAL = 100;

FileVersion: TYPE = LONG INTEGER;

ReadProperties: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, desiredProperties: LIST OF Property _ NIL, lock:
  LockOption _ [read, wait]] RETURNS [properties: LIST OF
  PropertyValuePair];
    --! LockFailed, Unknown {openFileHandle};
```

If *desiredProperties*=NIL, ReadProperties returns all file properties, ordered as in the declaration of Property; otherwise it returns the desired properties in the order requested.

```
WriteProperties: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, properties: LIST OF PropertyValuePair, lock: LockOption
  _ [update, wait]];
    --! AccessFailed {handleReadWrite, ownerCreate}, LockFailed,
      OperationFailed {unwritableProperty}, Unknown {openFileHandle,
      owner};
```

Writes all properties given in the *properties* list. The *type*, *immutable*, and *version* properties may not be written by WriteProperties (*type* is write once-only at file creation time; *immutable* may be set but not reset by MakeImmutable; and *version* is handled specially as described below). To write the *byteLength*, *stringName*, and *createTime* properties requires only that the *handle* be open with *access*=readWrite; to write *readAccess* and *modifyAccess* additionally requires that the client be the file's owner or a member of the create access control list for the file's owner; and to write *owner* additionally requires that the client be a member of the create access control list for the new *owner*.

```
ReadVersion: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle] RETURNS [version: FileVersion];
    --! LockFailed, Unknown {openFileHandle};

UnlockVersion: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle];
    --! Unknown {openFileHandle};
```

ReadVersion locks the version property in read mode. UnlockVersion releases a read lock on the version property. This allows the file to be updated by another transaction without conflict over the version. Note that if the current transaction has locked the file in read mode, that lock covers the version property, so it is useless to call UnlockVersion. For higher concurrency, the client must lock the file in intendRead, intendUpdate, or intendWrite mode and use UnlockVersion to unlock the version property after reading it.

```
IncrementVersion: PROCEDURE [conversation: ConversationID, handle:
  OpenFileHandle, increment: LONG CARDINAL];
    --! AccessFailed {handleReadWrite}, Unknown {openFileHandle};
```

Arranges that at transaction commit time *increment* will be added to the version property rather than 1. No update to the version property is visible, even to the transaction that incremented it, until commit time; if the transaction aborts, the increment is not performed.

5.7. Signals

The following are all the signals raised in AlpineFile:

```
AccessFailed: ERROR [missingAccess: NeededAccess];
NeededAccess: TYPE = {fileRead, fileModify, ownerCreate, handleReadWrite,
  spaceQuota};

LockFailed: ERROR [why: LockFailure];
LockFailure: TYPE = {conflict, deadlock, timeout};

OperationFailed: ERROR [why: OperationFailure];
OperationFailure: TYPE = {nonexistentFilePage, fileImmutable,
  unwritableProperty, tooManyRNames, stringTooLong};

Unknown: ERROR [what: UnknownType];
UnknownType: TYPE = {openFileHandle, volumeID, fileID, transID, owner};

PossiblyDamaged: SIGNAL;
```

6. Transactions and locks

All FileStore actions are carried out under *transactions* that ensure atomicity and consistency of updates in the face of concurrent requests from multiple clients, crashes of servers or clients, etc.

Briefly, a client first requests that some FileStore create a transaction; the FileStore subsequently serves as the *coordinator* of that transaction. If actions are to be carried out in other FileStores under the same transaction, the client must cause those FileStores to become *workers* for that transaction.

Reads and writes are performed under the transaction. The transaction machinery ensures that other clients will not see the state of the FileStore in a partially-updated (inconsistent) state; this is done by setting *locks* on files or parts of files, as discussed in section 6.2.

Eventually the client either *commits* or *aborts* the transaction. Committing causes all writes occurring under the transaction to be made permanently in the state of the FileStore and to be made visible to other transactions. Aborting causes all the writes to be abandoned, and makes the permanent FileStore state be as if those writes had never occurred. A transaction can also be aborted by a server or client crash or by a detected deadlock among transactions attempting to lock data in conflicting ways.

6.1. AlpineTransaction interface

All Mesa definitions in this section are declared in AlpineTransaction.

```
ConversationID: TYPE = RPC.ConversationID;

TransID: TYPE [9];

Create: PROCEDURE [conversation: ConversationID] RETURNS [trans:
  TransID];
```

Creates a new transaction, for which this FileStore is to be the coordinator.

A TransID may be treated as a capability, since it contains enough bits of unpredictable information to make it extremely difficult to forge. Therefore a transaction is shared among clients or private to the initiator according to whether or not the initiator hands the TransID to other clients. Alpine enforces no restrictions on who may present a particular TransID.

A FileStore on a workstation will not normally use its local file system to coordinate transactions involving multiple FileStores, since a crashed coordinator can tie up resources on other file systems. It is perfectly ok for a workstation's FileStore to coordinate its own transactions, or to act as a worker in a multiple file system transaction, since in these cases the only resources that it can tie up in a crash are its own.

```
FileStore: TYPE = BodyDefs.RName;

CreateWorker: PROCEDURE [conversation: ConversationID, trans: TransID,
  coordinator: FileStore];
    --! Unknown {coordinator, transID};
```

Informs the FileStore that it is to serve as a worker under an existing transaction *trans* which is being coordinated by *coordinator*. The client must call CreateWorker before performing any operations under *trans* on a FileStore that is not the transaction's coordinator. CreateWorker gives rise to some communication between the worker and the coordinator; Unknown[coordinator] is raised if the worker cannot locate or cannot contact the coordinator.

```
RequestedOutcome: TYPE = {abort, commit, commitAndContinue};
```

```
Outcome: TYPE = {abort, commit, unknown};

Finish: PROCEDURE [conversation: ConversationID, trans: TransID,
  requestedOutcome: RequestedOutcome] RETURNS [outcome: Outcome,
  newTrans: TransID];
    --! Unknown {transID};
```

Requests that transaction *trans* be finished in the specified way. This call must be made to the FileStore that is coordinating the transaction.

If transaction *trans* has already committed or aborted, Finish simply returns the outcome. Otherwise, Finish waits if necessary to assure a known outcome (which may differ from the one requested), and returns that outcome.

If *requestedOutcome*=commit (and *outcome*=commit), all actions previously requested are committed, all locks are released, and *trans* is terminated. If *requestedOutcome*=commitAndContinue (and *outcome*=commit), actions are likewise committed and *trans* terminated, but locks are downgraded rather than released, and a new transaction *newTrans* is created holding those locks. Locks are downgraded as follows: write, update, readIntendWrite, and readIntendUpdate to read; intendWrite and intendUpdate to intendRead. All OpenFileHandles referring to *trans* are changed to refer instead to *newTrans*, so the client may subsequently perform operations on existing open files under the new transaction.

The following are all the signals raised in AlpineTransaction:

```
Unknown: ERROR [what: UnknownType];
UnknownType: TYPE = {transID, coordinator};
```

6.2. Semantics of locks

This is a brief functional overview of Alpine file locks; for background material and further detail, see Alpine lock manager concepts''.

All locking is performed implicitly, as a side-effect of read and write operations on files and other objects. Consequently there is no public interface for dealing explicitly with locks; however, most operations provide optional means for a certain amount of client control over locks. Local clients, i.e., ones located on the same machine as the FileStore, *can* access the lock manager directly for the purpose of setting higher-level logical'' locks.

```
LockMode: TYPE = {read, update, write, intendRead, intendUpdate,
  intendWrite, readIntendUpdate, readIntendWrite};
```

When an object such as a file page is accessed, it is first locked in some mode. A read access must lock the object in read, update, or write mode, while a modify access must lock the object in update or write mode. Locking in update or write mode during a read access is sometimes useful when the client knows that it will later modify the object during the same transaction. The distinction between update and write modes will be explained shortly.

Once a lock is set on an object under some transaction, it will prevent certain types of access by any other transaction until the transaction holding the lock has terminated. The interaction between locks is defined by the function Compat[*r*, *e*], where *r* is a lock mode being requested and *e* is the mode of an existing lock on the same object by another transaction; it returns TRUE if *r* may be set immediately and FALSE if *r* may not be set until *e* has been removed.

Compat[read, read]=TRUE	Compat[read, update]=TRUE	Compat[read, write]=FALSE
Compat[update, read]=TRUE	Compat[update, update]=FALSE	Compat[update, write]=FALSE
Compat[write, read]=FALSE	Compat[write, update]=FALSE	Compat[write, write]=FALSE

The interaction between read and write locks is conventional: multiple readers or at most one writer (but not both) can coexist. An update lock is effectively a read lock at the time it is set but is converted automatically into a write lock at the time the transaction is committed (which is when any modifications to the locked object are logically performed); thus an update can proceed in parallel with any ongoing reads, but committing the update may require waiting until all read locks have been removed.

A transaction may *upgrade* one of its own locks by converting it to a stronger lock, with relative strength given by read < update < write. This is performed automatically, for example, when an object is read and later written during a single transaction. Of course, this upgrade may be blocked by locks set by other transactions.

Files are locked at two levels: the entire file may be locked in some mode, or pages, properties, etc., within the file may be individually locked. Locking the entire file substantially reduces the amount of work the server must do, since individual locks then need not be applied during each operation on the file; this is the appropriate style of locking for bulk file transfers and transactions involving private files. On the other hand, locking individual pages is required for shared data bases to maintain adequate concurrency.

The whole-file lock modes are read, update, and write, just as for other objects; these locks effectively cover'' individual operations whose modes are no stronger than the whole-file lock.

When pages of a file are to be locked individually as operations are performed, the entire file is locked in a weaker *intention* mode that specifies the strongest page lock expected; these modes are intendRead, intendUpdate, and intendWrite. For all *x*,*y*, Compat[intend*x*, intend*y*]=TRUE, but Compat[intend*x*, *y*]=Compat[*x*, *y*] and Compat[*x*, intend*y*]=Compat[*x*, *y*]. This enables detection of potential conflicts between page and file locks at the time the file is locked.

If an individual operation would require a lock stronger than the one implied by an intention-mode file lock, the file lock is automatically upgraded; this upgrade can of course be blocked by file locks set by other transactions.

Additionally, there are combination modes readIntendUpdate and readIntendWrite, which immediately lock the entire file in read mode with intention to perform individual operations requiring update or write locks.

Most operations have an optional *lockOption* argument, which permits the client to specify the lock mode to be used and the action to be taken if a conflict occurs; the default values of these arguments are appropriate for most applications. If *lockOption*.mode is too weak for the operation, it is ignored and the default used instead. If the lock cannot immediately be set and *lockOption*.ifConflict=fail, the operation immediately raises the error LockFailed[conflict]; otherwise the operation waits until the lock can be set, raising LockFailed[deadlock] or LockFailed[timeout] if a deadlock or timeout occurs. Not all deadlocks can immediately be detected as such; a lock timeout is treated as evidence of a deadlock. <The timeout interval is to be determined.>

7. FileStore global operations

The AlpineAccess interface contains procedures for querying the global state of the FileStore and for operating on the owner data base.

```
nullVolumeGroupID: VolumeGroupID = [...];

GetNextVolumeGroup: PROCEDURE [conversation: ConversationID,
  previousGroup: VolumeGroupID] RETURNS [group: VolumeGroupID, members:
  LIST OF VolumeID];
    --! Unknown {volumeGroupID};
```

Enables the client to enumerate the volume groups and volumes comprising a FileStore. If *previousGroup*=nullVolumeGroupID, GetNextVolumeGroup returns the first volume group; and when *previousGroup* is the last VolumeGroup in the FileStore, it returns nullVolumeGroupID.

<This description will surely evolve to include procedures for obtaining additional information about VolumeGroups and individual Volumes.>

Each VolumeGroup has associated with it an owner data base, as mentioned in section 4. The following procedures operate on the owner data base in various ways. All operations are performed under a transaction specified by the client. Many of the operations require that the client be a member of the Alpine wheels'' group, as discussed in section 4.2.

```
PageCount: TYPE = AlpineEnvironment.PageCount;

CreateOwner: PROCEDURE [conversation: ConversationID, trans: TransID,
  vol: VolumeGroupID, owner: OwnerName, spaceQuota: PageCount,
  createAccess, ownerAccess: AccessList];
    --! AccessFailed {alpineWheel}, LockFailed, OperationFailed
      {ownerAlreadyExists, ownerDataBaseFull, tooManyRNames}, Unknown
      {transID, volumeGroupID};

DestroyOwner: PROCEDURE [conversation: ConversationID, trans: TransID,
  vol: VolumeGroupID, owner: OwnerName];
    --! AccessFailed {alpineWheel}, LockFailed, OperationFailed
      {spaceInUseByThisOwner}, Unknown {owner, transID, volumeGroupID};

GetNextOwner: PROCEDURE [conversation: ConversationID, trans: TransID,
  vol: VolumeGroupID, previousOwner: OwnerName, lockDataBase: BOOLEAN _
  FALSE] RETURNS [owner: OwnerName, spaceQuota, spaceConsumed: PageCount,
  createAccess, ownerEntryAccess: AccessList];
    --! LockFailed, Unknown {volumeGroupID};
```

<*lockDataBase*=FALSE is of doubtful value, since a complete enumeration will read-lock the entire data base anyway.>

Each owner has associated with it two values relating to disk usage, *spaceQuota* and *spaceConsumed*, and two access control lists, *createAccess* and *ownerEntryAccess*. Members of an owner's *createAccess* list are permitted to create files whose disk space is charged to that owner. Members of *ownerEntryAccess* are permitted to make changes to the owner entry itself.

```
OwnerProperty: TYPE = {space, createAccess, ownerEntryAccess};
OwnerPropertyValuePair: TYPE = RECORD [
  SELECT property: OwnerProperty FROM
    space => [spaceQuota, spaceConsumed: PageCount],
    createAccess => [createAccess: AccessList],
    ownerEntryAccess => [ownerEntryAccess: AccessList],
    ENDCASE];


ReadOwnerProperties: PROCEDURE [conversation: ConversationID, trans:
  TransactionID, vol: VolumeGroupID, owner: OwnerName, desiredProperties:
  LIST OF OwnerProperty _ NIL] RETURNS [properties: LIST OF
  OwnerPropertyValuePair];
    --! LockFailed, Unknown {owner, transID, volumeGroupID};
```

If *desiredProperties*=NIL, ReadOwnerProperties returns all properties for *owner*, ordered as in the declaration of OwnerProperty; otherwise it returns the desired properties in the order requested.

```
WriteOwnerProperties: PROCEDURE [conversation: ConversationID, trans:
  TransactionID, group: VolumeGroupID, owner: OwnerName, properties: LIST
  OF OwnerPropertyValuePair];
    --! AccessFailed {alpineWheel, ownerEntry}, LockFailed,
      OperationFailed {tooManyRNames}, Unknown {owner, transID,
      volumeGroupID};
```

Writes all properties given in the *properties* list. In the case of the *space* property, only *spaceQuota* is written and *spaceConsumed* is ignored. To write *ownerCreate* requires that the client be a member of the file's *ownerEntryAccess* list; to write *space* or *ownerEntryAccess* requires that the client be an Alpine wheel.

The following are all the signals raised by the procedures described in this section:

```
AccessFailed: ERROR [missingAccess: NeededAccess];
NeededAccess: TYPE = {alpineWheel, ownerEntry};

LockFailed: ERROR [failure: LockFailure];
LockFailure: TYPE = {..., deadlock, timeout};

OperationFailed: ERROR [why: OperationFailure];
OperationFailure: TYPE = {ownerAlreadyExists, ownerDataBaseFull,
  tooManyRNames, spaceInUseByThisOwner};

Unknown: ERROR [what: UnknownType];
UnknownType: TYPE = {owner, transID, volumeGroupID};
```

References

[Alpine, 1981]
Alpine documentation is kept on-line in the [Ivy]<Alpine>doc> directory. In the file names, *'' stands for a number; successive releases of a document are assigned the next higher number.

Alpine file server overview'', AlpineOverview.press

Alpine lock manager concepts'', LockConcepts*.bravo

FileStore interface internals'', FileStoreInternal*.bravo

[Needham & Schroeder, 1978]
Roger M. Needham and Michael D. Schroeder, Using encryption for Authentication in Large Networks of Computers'', *Communications of the ACM*, vol. 21 no. 12, December 1978.

Change history

Version 8; October 13, 1981 5:29 PM. Add AlpineFile.Close, LockPages, UnlockPages. Finishing a transaction with commitAndContinue now returns a new TransID rather than permitting additional operations under the old transaction. Substantially simplify interface to RPC runtime machinery.

Version 7; September 21, 1981 8:38 AM. Further refine to conform with RPC design. Add high water mark mechanism. Delete the individual property read/write procedures. Document semantics of locks. Change LockMode to LockOption. Change to read/write owner entry in same style as file properties.

Version 6; September 14, 1981 8:46 AM. Change style of this memo to make it serve better as client programmer's documentation for the FileStore public interface; remove internal details (implementation strategies and the like) to a separate memo, FileStore interface internals''. Split the former FileStore interface into four pieces: AlpineEnvironment, AlpineAccess, AlpineFile, and

AlpineTransaction. Bring conversation initiation and authentication into conformity with the current RPC design. Make a first cut at specifying the signals.

Version 5; August 25, 1981 12:41 PM. Make interface handle-oriented, since RPC requires it; see Handle, Create, Destroy. A Handle corresponds to a client talking to a *server*, not to a volume or a machine; some procedures take a volume (or volume group) parameter. Make the identification FileStore = log, and introduce a separate notion of logical disk volume and volume group. Add no-logging option to Open/CreateFile; this takes the place of noTransaction, since it handles the one situation in which we can see that clients will want to give up recovery to reduce the size of the log (large-scale replication of files, where the home copy is kept logged and other copies are unlogged.) OpenFileID -> OpenFileHandle, AccessListID -> AccessList (a list of RNames.) Add section on owner database (CreateOwner, ... , DestroyOwner), and expanded file properties section to include all properties we now plan to support.

Version 4; May 29, 1981 5:36 PM. Introduced notion of file owner, who is charged/credited for pages used/released in Create/DeleteFile and SetLength.

Version 3; May 15, 1981 11:02 AM. Introduced types OpenFileID, ClientID, AccessListID, ... Added noTransaction. More detail on file opening and creation; support file create with and without an externally supplied ID (used to be only with.) Length -> ByteLength throughout. List-oriented file property operations. Much more work required on property interface, including locking issues.

Version 2; March 12, 1981 11:22 AM. Added more detail to the Files'' section, including procedures to manipulate file attributes. Made lock modes consistent with LockDesign0. Made minor changes to Transaction section: RegisterWorker takes a restart ID'' to allow unilateral abort of worker; FinishTransaction and FinishWorker take a requiredOutcome that may be commitAndContinue or commitAndTerminate; expanded comments on Transaction section operations.

Version 1; February 25, 1981 3:54 PM. Lock stuff moved out (to LockConcepts0.bravo, LockDesign0.bravo.) Sierra -> Alpine.

Version 0; February 19, 1981 3:16 PM. Interface name changed from BFS''. We are attempting to erase the distinction between the regular interface and on the wire''. The volatile structures Coordinator'', Worker'', and File'' have disappeared. The type LockID has been defined, following Ed Taft's suggestion.

Unfinished business

Read ahead / write behind hints in the interface. Be sure that fast streams can be implemented on top of FileStore.

Log/recovery interface.

Cache registration (at what level should this be done?)

Lock interface (no conceptual difficulty, details can be deferred to implementation time.)

There used to be a CoordinatorFileSystemIDFromTransID operation, which is now hard to implement given that FileStores are identified by RNames rather than UniqueIDs. How necessary is this operation?

ContentID property?

Indefinitely deferred business

Validation of remote page caches (through a history database that is coordinated with file page updates.)

Transaction save points?