The File System Architecture of OSF/1 AD Version 2

Paul Roy
David Noveck
Durriya Netterwala

OSF Research Institute
{roy,dnoveck,durriya}@osf.org

August 23, 1993

1 Introduction

Between 1991 and 1993, the OSF Research Institute developed the OSF/1 AD operating system for multicomputers [5] [9]. Version 1 of OSF/1 AD became operational in 1992 and has served as the basis for Paragon OS as shipped on the Intel Paragon. Version 1 is also the basis for other vendors' technologies.

Presently, the OSF Research Institute is engaged in the design of OSF/1 AD Version 2. This version will include enhancements in the areas of cross-node Mach IPC, server co-location in the system address space, and a new vnode-based file system architecture.

As in Version 1, the Version 2 file system will support very high bandwidth file I/O, and will provide transparent access to remote files and devices by implementing a coherent system-wide file name space. The file system will also allow file system information to be cached on multiple nodes, and will maintain the coherency of all cached information. A complete set of requirements and goals of the Version 2 file system is presented in [6].

After providing an overview of OSF/1 AD, this document describes at a high level the new file system architecture of OSF/1 AD Version 2. More detailed information will become available in the future.

2 OSF/1 AD - A Multicomputer Version of OSF/1

OSF/1 AD extends OSF/1 to a multicomputer environment, providing a full *single system image*. Its architecture is derived directly from the target hardware architecture, in which large numbers of nodes are connected via a high-speed interconnect, and some subset of the nodes have peripheral devices attached. OSF/1 AD must allow all nodes and devices to be efficiently utilized and, in particular, must distribute OS functionality to avoid bottlenecks.

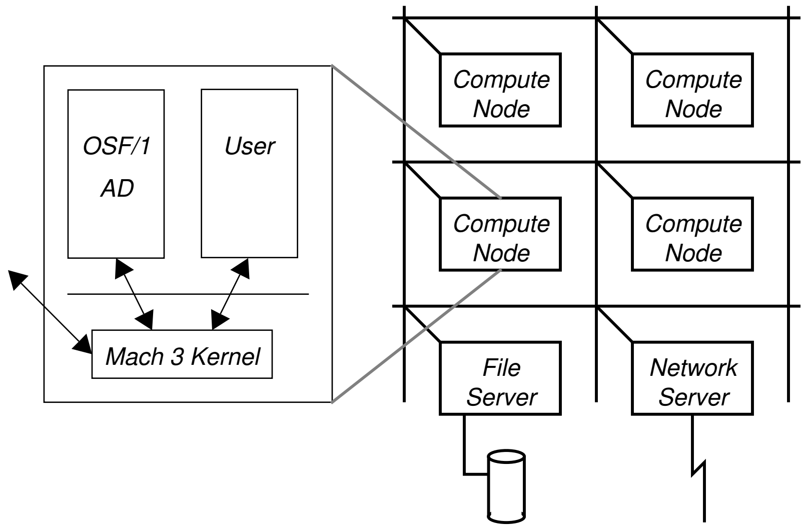

Figure 1: OSF/1 AD Version 2 Architecture

These considerations lead to the following model for the Version 2 architecture (see Figure 1):

- The Mach kernel runs on all nodes of the multicomputer, providing generic task/thread management, memory management, communication services, and device access.
- Specialized user-space servers implement Unix functionality, such as file service, process management, and networking.
- Disk and networking devices are managed by servers which are typically co-located on the same node with the device, although co-location is not mandatory.
- Each application has a single server responsible for receiving all its system calls. This server is referred to as a *system call agent*. The binding of applications to system call agents, and the binding of system call agents to nodes is flexible, but typically, all applications running on a node will use a local system call agent.
- Process management functionality is distributed across system call agents. Optimally, process management system calls may be handled entirely by a local agent, although some calls may require remote interaction.
- File management system calls are handled by an applications's system call agent. This agent provides access to remotely managed files through a vnode-based distributed file system.

OSF/1 AD implements the entire OSF/1 Application Programming Interface (API). Although OSF/1 AD assumes no physical shared memory between nodes, it supports shared memory between processes residing on different nodes (via the XMM subsystem). Transparent interprocess communication is provided by an extended Mach IPC to support cross-node communication (NORMA IPC).

The major differences between Versions 1 and 2 of OSF/1 AD include the absence of an emulator in each process's address space, the use of vnode technology to implement the distributed file system, and the system's ability to co-locate a server in the kernel address space [1]. The latter optimizes the interaction between application and system call agent during system call execution.

3 File System Architecture of OSF/1 AD Version 2

The Version 2 file system will be distributed across all nodes in the multicomputer, and will rely on VFS (Virtual File System) technology in its implementation. File servers will implement the server-side of the file system and system call agents will implement the client-side.

The remainder of the document discusses the particular aspects of the file system architecture.

3.1 File Name Space

As in OSF/1 AD Version 1, Version 2 extends the standard OSF/1 file system to support the integration of multiple file servers into a coherent whole. Each file server provides file service for one or more file systems (partitions), and mount operations are used to assemble multiple file systems across multiple file servers into a single file name space with a single root directory (Figure 2).

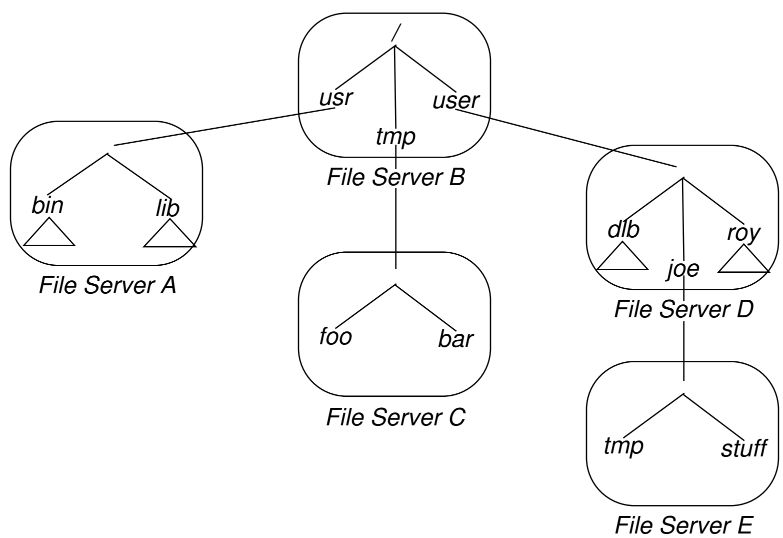

Figure 2: File Name Space

3.2 Object Interposition

Interposition is a structuring mechanism allowing file system functionality to be provided in a modular fashion. In OSF/1 AD Version 2, interposition will be used for both vnodes and mount structures. This discussion only motivates the use of interposition for vnodes; the motivation for mount structures is analogous. See [3] for more information on the interposition mechanism.

In standard OSF/1, a file is represented by a vnode (its identity) and is implemented by a file system-dependent structure providing its behavior. Vnode interposition allows behavior to be added to a file system by associating multiple file system-dependent structures with a single vnode. These structures are known as *pvnodes* and are linked together into *interposition chains*. Much like the existing vnode structure, each pvnode has an associated operations vector.

In OSF/1 AD Version 2, interposition will be used to cleanly add behavior to a file, such as maintaining coherency in the face of distributed, cached access by multiple client-side servers. The intent is to improve the situation in OSF/1 where presently the best way to add file behavior is to "steal" the vnode operations vector (e.g., as is done in DCE DFS). It is similar in concept to the stackable file systems work currently fashionable in the file system community [7] [8], but not as ambitious because it's not a goal to support stacking new, previously unknown file systems at run-time, or to support absolute binary compatibility with existing loadable file systems.

Figure 3 shows how a pvnode may be interposed on an UFS inode to allow distributed access.

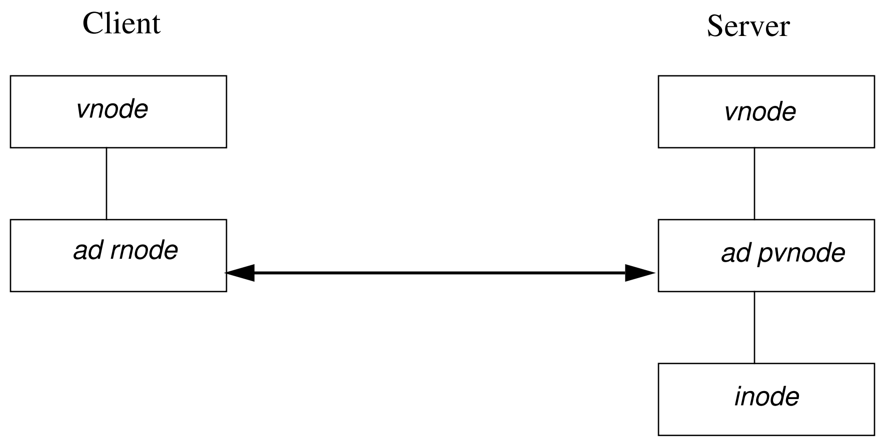

Figure 3: Object Interposition

3.3 Token Module

The *token module* provides services to allow multiple servers to coordinate their activities. Tokens, by convention among those dealing with a given object, grant the right to make certain changes in the state of an object, or prevent others from making such changes, thereby allowing safe read-only caching of data about the object.

Tokens are granted for individual objects. Each such object has an object manager within a given server and one or more client object users within other servers. Object managers and object users communicate directly, using the token module to manage their respective token-related state.

Tokens for a given object are divided into a set of classes, each class designating some part of the state associated with an object. Tokens of different classes for a given object do not conflict. For a given class, there are tokens of a number of levels available. Tokens of different levels for a given class do conflict; only one level may be outstanding at a given time. The read level allows shared access in that many instances of the same token may be given out at the same time. The write level only allows a single token to be outstanding. In addition to the read level, another shared level allows multiple coordinated updates to proceed. This requires that the object manager implement a consistent rule to resolve multiple updated values of a data item. This level of token will be used by pvnode objects to allow multiple readers all with the right to update the access time associated with a file. Multiple updates will be resolved by server-side code by only allowing the time to increase monotonically when this level of access to the access-time token class is in effect.

OSF/1 AD Version 2 will use tokens for several different objects including mount structures, file structures, and vnodes. The token module will provide the mechanism for acquiring, releasing, and revoking tokens, including batched operations (e.g., acquiring multiple tokens at once). The token module itself is divided into client and server pieces which are co-located in the same address space as the object's client and server-side pieces, respectively. No aspect of the token module has knowledge of object-specific semantics. Interaction between object-specific code and token code is through well-defined interfaces. Any token operation that may require remote interaction, such as acquiring or releasing tokens, is initiated by having the token module invoke object-specific code, allowing piggybacking with object-specific operations. Thus, for example, attributes may be obtained and cached for future use at the same time a file's attribute token is acquired, or modified state associated with a token may be written back to the object's server when the token is released.

For a file, the notion of allowing conflicting tokens for different pieces, or ranges, of the file supports greater concurrency and greater cacheability. Such an enhancement is not planned for Version 2.

Figure 4 shows the token module and its relationship to the pvnode for a file being accessed remotely. Note that there is no direct communication between the token module's client and server code.

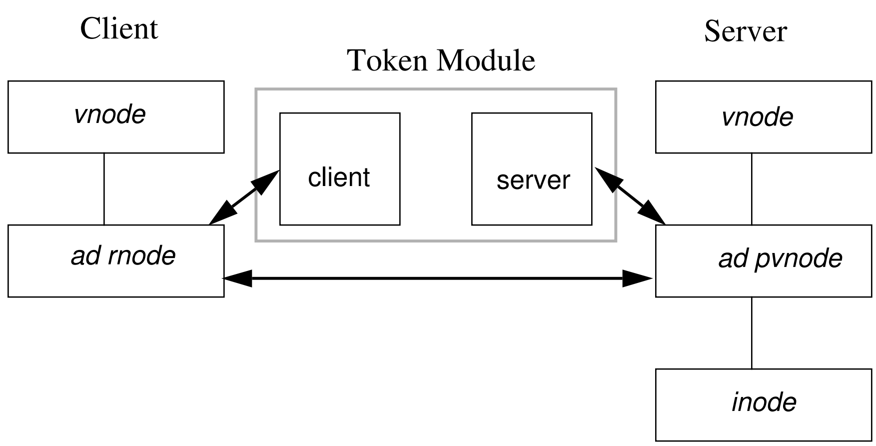

Figure 4: Token Module

3.4 Remote Object Identifiers

All remotely accessible objects must have a unique identifier. Because these identifiers are heavily used throughout the system, their structure and management are important aspects of the architecture. The requirements for such identifiers are:

- cheap creation
- low memory consumption
- on client-side, cheap conversion from an id to the corresponding target Mach port
- on server-side, cheap conversion from an id to the corresponding object
- cheap transfer across address spaces and nodes
- ability to cheaply implement cross-node referencing counting to the corresponding object
- identical id's for a given object within a single address space

In addition, in some cases it's useful to have an object's id be identical system-wide (across address spaces) because back references to a large number of clients with access to the object may be compactly encoded (e.g., a range of clients may be encoded in a single word).

In Version 1, most remote object identifiers are Mach ports. Mach ports allow an object's server to transparently change (by moving the port's receive rights), but have the disadvantage of being fairly costly.

For these reasons, Version 2 will reference remote objects by opaque *handles*. Handles are object-specific in terms of structure, conversion to and from ports, and transfer across address spaces. This architecture is flexible enough to accommodate lightweight handles such as uids, or heavier weight handles providing greater functionality such as Mach ports.

Most handles will be (server id, object id) tuples. For example, the handle for a remote vnode will consist of a 32 bit server id and 64 bit object id, the latter containing 32 bits for the pvnode address and 32 bits for a unique generation number. A server id will map one-to-one with a server port.

3.5 Referencing Remote Objects

The architecture supports both hard and soft remote references. A hard reference is a true reference to the corresponding object. For example, a client with a file open will have a hard reference to the corresponding server-side vnode.

A soft reference is simply a guarantee to the possessing client that the server-side object will not disappear. Because soft references do not hold true object references, they are useful for allowing server-side objects to be independently reclaimed (i.e., the reference count may go to zero). Thus, for example, a client that has a file's attributes cached, but doesn't have the file open, will have a soft reference to the server-side vnode. If the server needs to reclaim this vnode, say because of heavy demand from other clients, it may attempt to "break" all outstanding soft references. Breaking a client's soft reference causes it to relinquish its access to the corresponding object.

A client may refuse to allow a soft reference to be broken, in which case the reference acts like a hard reference because it prevents the corresponding object from disappearing. Such a situation is conceivable when, say, a client holds a soft reference and would like to acquire a hard reference, but defers the acquisition in order to avoid the cost of a RPC. Deferring the acquisition in this way enables optimizations such as completely avoiding the RPC (if the hard reference is subsequently relinquished), or batching, in which case multiple deferred operations are executed as part of a single RPC. Similarly, relinquishing a hard reference may also be deferred.

Soft references are supported through the token module, via *existence* tokens. A client acquires a soft reference by acquiring an existence token for the object. A server may attempt to break the soft reference by attempting to revoke the existence token. If the client rejects the revoke, then it has effectively prevented the soft reference from being broken.

3.6 Multiple Remote File System Types

OSF/1 AD Version 2 supports multiple types of remote file systems within its name space. A *file system type* in this context refers to a well-defined behavior, or particular collection of behaviors, exhibited by a set of "files." A file's behavior may be determined by its type (i.e., the operations it supports) as well as its implementation-oriented behavior (such as its performance under varying workloads).

In practice, different file behaviors may be provided by different client-side and/or server-side file system-dependent vnode structures (pvnodes) and associated operations vectors. The type of pvnode associated with a vnode may depend on the corresponding file's type (e.g., plain file vs. device special file vs. fifo) as well as mount options.

The following types of remote file systems will be provided by the OSF Research Institute.

Caching File System. This file system will provide client-side caching of pathnames, attributes, and file data. Pathnames will be cached using the standard OSF/1 directory name cache. Cached attributes will be associated with client-side vnodes, each of which must at least hold a soft reference (existence token) to the corresponding server-side vnode. File data caching will be provided through the use of Mach memory objects, as described in Section 3.7.

The cached file system will aggressively piggyback token operations with remote file system operations. In addition, open and close operations will be deferred when possible to avoid RPCs. Actual execution of deferred operations may be piggybacked on subsequent RPCs, including those carried out on behalf of the sync() system call.

Fast Path File System. This file system is similar to the cached file system except file data is not cached. This enables very high bandwidth transfers because data caching mechanisms are avoided. Optionally, pathname and/or attribute caching may be turned off as well. A mount option will be used to indicate whether files should exhibit fast path or full caching behavior. Vendors will likely enhance this file system as necessary to provide additional high-performance characteristics (e.g., file striping, different read-ahead strategies, different synchronization strategies, etc.).

Device File System. This file system supports access to remote devices. In particular, it must support all combinations of the following entities residing on the same or remote node(s): an application accessing a device, the name for the device (its device special file), and the physical location of a device. And, it must be efficient in that reads and writes of an open device must not involve intermediate nodes. Finally, it must support full Unix semantics, including proper tracking of a device's accessed and modified times.

This functionality will be provided by using interposition in the client-side server. An upper pvnode will be responsible for a device's attributes, including proper tracking of accessed and modified times and subsequent write-back of this state to the corresponding device special file. A lower pvnode will be responsible for reading and writing the device's data. Both the attribute and data pvnodes will come in two flavors, one each for the corresponding device special file or device itself being local or remote.

Fifo File System. Fifos (aka named pipes) are accessible through the file system name space. The fifo file system provides access to an open fifo by directing vnode operations to its corresponding file server, which in turn may invoke resident networking code. Alternatively, the fifo file system may choose to invoke networking code directly from the client-side if that code is available.

Import File System. File systems provided by external NFS or AFS servers may be mounted into AD's file name space. The import file system supports access to such files by simply connecting the client-side to a server-side running client code for the external file system (e.g., NFS or AFS client code). The import file system will not cache file system information on the client-side, nor will it attempt to take advantage of local hardware connected to an external network and short-circuit the server-side when communicating to an external file system server, although such optimizations are possible.

Note that the implementation of these remote file systems will involve a great deal of code sharing. For example, the client-side pvnode and associated operations vector will be largely the same for the cached file system, fast path file system, and remote flavor of a device's data pvnode. The main difference will be the read and write operations; the cached file system performs data caching, the others don't. In addition, the directory-related operations will not be valid for the device's data pvnode because it's not possible to have such a pvnode for directories.

3.7 Data Caching

The cached file system will support caching of file data through the use of Mach memory objects. The mechanisms will be similar to those in Version 1, as reported in [5], but access to memory object data on behalf of read and write system calls will take place in a client-side server as opposed to a per-process emulator. Version 2 will also have a better structured implementation because caching support will be encapsulated in the interposed pvnode on the server-side, rather than being added to the underlying UFS.

The OSF Research Institute will improve support for memory object based Unix files by enhancing Mach's architecture as follows:

- New VM interfaces for direct access to memory object data. These interfaces will allow the throughput to scale with the size of a request (not one fault per page), provide a channel through which useful information can be passed (such as a hint to allow the kernel to avoid needless interactions with the pager when growing a file), and allow error codes to be easily returned from Mach to the invoking code.
- Enhancements to the External Memory Management Interface (EMMI) to allow memory objects to be used *without* XMM's multi-node cache coherency mechanism. As multi-node data coherency will be provided by the Unix file system layer, this enhancement allows unnecessary XMM machinery to be avoided.
- Improved Mach kernel page management. The Mach kernel will be modified to better account for the competing demands of memory pages due to file data, program data, anonymous data, etc.

3.8 Local File System

OSF/1 AD Version 2 will provide local file system service using an enhanced UFS. Third-parties may add additional local file systems with relative ease, however. These enhancements include:

- No-buffer-cache for file data. A streamlined data path that eliminates reuse caching and data copies will be provided. It will be used by both the fast path file system and the cached file system.
- Disk block reservation. The UFS will export a new vnode interface allowing disk blocks to be reserved for files. This interface is required to support memory object-based caching, and hence represents a required modification to other potential local file systems. Note that this interface can serve as the basis for a new system call interface allowing applications to preallocate disk blocks for files.
- Extent-like performance. The UFS will support large contiguous disk transfers through the use of contiguous disk block allocation, and disk block coalescing at read and write time.

The first two of these enhancements are also present in OSF/1 AD Version 1.

3.9 Large Files

OSF/1 AD Version 2 will support 1 Terabyte files for systems that provide 64 bit long and off_t data types. The 1 Tb limitation comes from UFS's use of 32 bit disk addresses, where each disk address is assumed to reference 512 bytes of data. With relatively little work, the code base could be modified to properly distinguish sector addresses from fragment addresses, thus allowing the maximum file size to be increased by an additional factor ranging from 2 to 16 depending on how the on-disk file system is configured (assuming addressable fragment sizes ranging from 1024 to 8192, respectively).

3.10 Mount Model

As described in Section 3.1, OSF/1 AD has a single file name space with a single root directory. This section provides an overview of the model for setting up the name space. A separate document provides a detailed description of the mount mechanism [2].

The mount model in OSF/1 AD is identical to standard OSF/1. The mount command is used to graft a self-contained directory hierarchy (supplied by either a file system partition or external server), onto a particular directory in an already-existing hierarchy. The root directory must be mounted in a "bootstrap" fashion because there is not an already-existing directory hierarchy.

When a file system partition is mounted in standard OSF/1, both the mounted-on directory (the *covered vnode*) and partition may be assumed to be locally managed. In OSF/1 AD, however, either or both of entities may be managed by servers remote from where the mount command is executing. In addition, the server for the device's name (its special file), may be remote. (Note that, as in Version 1, the server for a device may be determined from a node number stored in the device's special file.)

OSF/1 AD Version 2 will modify the file system-independent mount code to break the assumptions that the covered vnode and file system partition are locally managed. This will be supported in part by a new VOP_DOMOUNT operation for directories which does the work of "covering" a directory with a (potentially remote) file system partition. In the case of a covered vnode and the partition being managed by distinct file servers, there will exist both client-side and server-side mount structures (for the covered vnode and partition, respectively) through which the connection is maintained.

As for the root file system, it will be mounted locally by a *root file server*, and then imported at bootstrap by system call agents and other file servers. These servers are able to determine the location of the root file server through configuration information supplied at bootstrap.

Another aspect of the mount model, adapted from [4], is the ability for a system call agent to lazily evaluate a copy of the covered vnode's mount structure when it is encountered during pathname traversal. The system call agent is then able to connect directly to the server-side mount structure, thus avoiding interaction with the server managing the covered vnode during subsequent pathname traversals.

Finally, the mount mechanism will be adaptable to a directory-on-directory mount model. Such a model might be desirable in some cluster environments where each node has its own local name space, and directory-on-directory mounts are used to graft the name spaces together. See [2] for the details of the actual mechanism.

3.11 Process Management Considerations

The file system and process management functionality have various dependencies on each other. Client-side file system code depends on process management code in the system call agent to provide:

- An execution environment, including a thread with associated uarea/proc structure containing at least the process's credentials and file resource limits.

- A signal posting service, so that signals may be posted (e.g., SIGXFZ in the event that EFBIG is returned from a remote file server).

Similarly, file servers also need an execution environment upon receiving a client-side request, along with the ability to set the environment's credentials and resource limits based upon those received in the request message.

Process management also has dependencies on file system code for the following:

- Registering in support of interruptible system calls. Prior to sending an RPC to a remote servers, file system code must register enough state with process management to enable interruption of the remote operation on behalf signal processing.
- exec() depends on the file system to resolve the pathname of the file being executed, which is possible because exec() and file system code (client-side or server-side, depending on whether the system call agent is also a file server) will be co-located in the same server. However, exec() also requires the file's memory object port (so that it may map it), but the port is only obtainable locally in standard OSF/1. This will be rectified by adding a new vnode operation, VOP_PAGER_SETUP, to obtain the memory object port through the file system.
- rfork() causes a child process to be bound to a system call agent potentially distinct from the parent's. This requires a file system mechanism to duplicate the parent's file system information in the new agent (including open file state and the root and current working directories), and to set up appropriate sharing relationships. This mechanism will partly be supplied by new FOP_GETINFO and FOP_SETINFO calls which will allow an open file's state to be duplicated appropriately, regardless of whether the file is vnode-based or socket-based.

  For vnode-based files, the token module will be used to facilitate caching of a file's offset between file structures in different servers. By default, the server for an ancestor process will contain the token server functionality and the server(s) for descendents will contain token client functionality. If the file is closed for the last time in the ancestor server, then a new token server will be designated from the set of existing token clients.
- Process migration. The file system mechanisms in support of migration are similar to those provided in support of rfork() because the migration operation is equivalent to creating a child with access to the parent's open files, and then closing the files on the parent.

3.12 Networking Considerations

In addition to providing a name space for fifos, sockets, and STREAMS devices, the file system must support the passing of file descriptors from one process to another. The mechanism used will be similar to that used to setup the child's open file table during rfork() processing.

3.13 Minimal Client-Side File System

In support of applications that require absolute minimal file system functionality (as a way of reducing OS memory consumption), OSF/1 AD Version 2 will be configurable through the use of compilation switches. The ability to configure-out unneeded file systems, including the caching file system, as well as other functionality such as the buffer cache, will be provided.

3.14 Failure Containment

Failures will be contained in that unrelated parts of the file system name space will be unaffected when a file server fails. Also, all object references associated with failed servers will be relinquished, thus allowing appropriate cleanup and object reclamation. The mechanism for relinquishing such references will be based on dead-name notifications from Mach ports. To minimize the overhead involved with requesting and setting up notifications, they will be requested at the server granularity when possible, augmented by internal data structures to associate objects with a particular server.

Thus, for example, client-side code caching files from a remote file server will have vnodes for those files, each one associated with a local mount structure, which in turn is associated with a remote mount structure. The client-side code will register its local mount structure internally by associating it with the file server for the remote mount structure. If the remote server fails, resulting in a dead-name notification, the client-side code may clean up the associated mount structure which in turn enables it to cleanup all the cached vnodes from that mounted file system.

If a system call agent fails, file servers will use a similar mechanism to free up references and release tokens that the agent may have held. If a failed agent is itself a client of another agent (because of shared file structures), the other agent will be able to cleanup similarly. A more difficult problem occurs if a failed agent is the server for a shared file structure. In such cases, the open file will become inaccessible to all other client agents of this file.

In general, failures may invalidate user assumptions, particularly when the system is using high-performance caching techniques. For example, failures can lose state such as file data written to cache only, or a modified offset associated with an open file. If necessary, it's possible to trade-off performance to mitigate the effect of such scenarios.

4 Acknowledgments

Jose Rogado of the OSF Grenoble Research Institute provided useful input, through discussions and a file system prototype. Bill Bryant helped improve the readability of this document.

References

[1] Condict, M., Mitchell, D. Optimizing Mach-based Systems By Server Co-location. OSF Research Institute internal document. August 1993.

[2] Netterwala, D. The Mount Mechanism in OSF/1 AD Version 2. OSF Research Institute internal document. August 1993.

[3] Noveck, D. Interposition Mechanism in the OSF/1 AD Version 2 File System. OSF Research Institute internal document. August 1993.

[4] Rogado, J. A Prototype File System for the Cluster OS. OSF Research Institute internal document. May 1993.

[5] Roy, P. Unix File Access and Caching in a Multicomputer Environment. In *Proceedings of the USENIX Mach Symposium*, April 1993.

[6] Roy, P., Noveck, D., Netterwala, D. OSF/1 AD Version 2 File System Requirements. OSF Research Institute internal document.

[7] Skinner, G., Wong, T. "Stacking" Vnodes: A Progress Report. In *Proceedings of the Summer 1993 USENIX Conference*.

[8] Unix International, Stackable Files Work Group. Requirements for Stackable Files. Revision 3.6, February, 1993.

[9] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., Netterwala, D. An OSF/1 Unix for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*.