A Prototype File System

for

the Cluster OS

José Rogado

OSF Research Institute
2, Ave. de Vignate
38610 Gières
France
Tel: (+33) 76 63 48 66
E-mail: jose@gr.osf.org

Abstract:

This document presents a framework for building a Cluster File System based on NORMA Mach IPC, using the standard OSF/1 internal *vnode* and VFS interfaces. Most of the concepts used in this prototype were already discussed in a previous document[1]. Therefore, it is not the aim of this paper to present a full solution for a complete Cluster File System. On the other hand, except where explicitly mentioned, all the functionality described in this paper is implemented and tested under prototype conditions.

1. Introduction

As a consequence of the so called "Cluster Project", a certain number of experiments and concept validations have been achieved in Grenoble during the last six months. Those lead to the design and implementation of a prototype Distributed File System, on which some of the previous AD File System ideas where used, together with a more layered approach of the system's architecture.

This document describes the main features and gives implementations details of what we first called the "Port[2] File System" (PFS), which by no means we pretend to be a complete Cluster File System. On the contrary, we consider PFS as a framework on which complementary technology acquired from third parties, as well as some of the current AD functionality, can be included.

Next section gives a general overview of PFS main reasoning and design options, while the following ones present implementations issues in more detail.

---

1. A Strawman Proposal for the Cluster Project, October 1992
2. Or Prototype File System.

2. PFS rationale and overview

In the current version of the AD File System (ADFS), the emulator is used as a repository for file state. The OSF/1 file table is in fact kept in it, and opened files are represented by Mach ports. This way, once a file is opened, all the subsequent file operations are directed to the server actually handling the file. On the other hand, this approach prevents the use of traditional file attribute and pathname caches without porting parts of them to the emulator.

The vnode interface

Recently, the OSF/1 MK system evolved to a no-emulator architecture, and the AD system is also going to adopt this direction in its next version. Consequently, the AD File System will no longer be able to rely on optimizations that depend on keeping file state in the emulator. In such an architecture, all the system calls directed to files will be handled by a server, and consequently a more traditional approach is needed. The *vnode* and VFS interfaces, already present in the OFS/1 code basis, are natural solutions for handling different file system types in a layered and widely used fashion. So, part of the work we achieved was to extract the main ideas that were behind ADFS and insert them in a new file system type accessed through the *vnode* and VFS interfaces.

Multi-component lookup

One of the most important optimizations of ADFS, in respect to more traditional file systems, relies on its ability to forward pathname lookup operations to the server handling the remaining part of the pathname. This operation is performed by the root server, or any other file server, upon detection of a remote mount point during pathname translation.

The way the lookup operation is performed by the standard *vnode* layer is one component at a time. This would highly increment the number of messages needed for pathname resolution, and would have a negative impact on system performance. So our approach consisted also in developing a multi-component lookup under the *vnode* layer.

Global mountpoints

In ADFS, mount points are global, and this is a key feature in providing a single system image. On the other hand, traditional VFS mount points are created and solved by the client side, and therefore cannot be exported. One of the most important parts of the work done in PFS consisted in making VFS mount points global, from the creation and resolution point of views. In fact, mount points are always created on the server handling the directory that is being covered. This, together with a lazy evaluation caching and revocation scheme provides a way for creating a single system image using the traditional VFS interface.

Attribute caching and revocation

Together with the *vnode* abstraction comes the possibility of handling a cache for file attributes. In fact, the file system dependent part of the *vnode* can be used for storing the attributes of the file the *vnode* represents. In other distributed file systems like NFS, this cache coherency is kept based on time stamps. In AFS, a special token mechanism is used to invalidate the file attributes. In PFS, we decided to implement a revocation mechanism based on Mach port rights, which in fact is the same we use to keep cached mount points coherent.

Pathname caching and revocation

Although this functionality is not yet present in PFS, it is our intention to implement pathname

caching, which will be based on the current VFS caching primitives. Those will be extended to allow multiple component chaining, in order to cope with the new type of lookup introduced. To minimize memory usage, entries with common heading pathnames will share the same storage locations in the pathname cache, which will thus be organized as a hierarchical tree. Entries effectively stored in the cache will be revoked nominally, while intermediate component revocation will need a full tree invalidation.

Mapped File Access

The current version of ADFS relies on Mach VM to cache data when accessing files. Although this implementation is not architectured around the *vnode* and VFS layers, we believe from discussions with its implementor[1], that it will be relatively easy to adapt it to this model. This will provide a means for caching remote and local files, although a protocol providing a looser coherency scheme than the one provided by NORMA XMM, needs to be implemented to avoid page bouncing on concurrent accesses.

In the next sections we cover in more detail the relevant PFS functionalities.

3. The PFS mount functionality

In order to access a file system through PFS, the distant file tree has to be mounted using the standard OSF/1 mount command, which has been extended to consider PFS as another file system type. PFS mount points are thus created by commands of the type:

mount [-t pfs] <node #>_<remotedir> <localdir>

where <node #> is an integer which identifies a node in a NORMA domain. This syntax also means that the remote directory has to be part of the remote node file tree before been accessed by a client. Note that since mount associates the "_" with the PFS type, the [-t pfs] argument is optional.

3.1 PFS mount points

PFS mount points can be created locally or remotely, i.e. on file tree already accessed through PFS.

Local mount points

A local mount point creates an entry on the client mount table. A send right for the server remote directory is kept on the file system dependent part of this entry. After a local PFS mount point is created, all subsequent accesses to <localdir> and beyond will translate into VFS, and *vnode* operations are forwarded to the remote node server using Mach IPC.

A PFS mount point gives visibility to the server file tree as a whole, including other exported non-PFS mount points, which are crossed by the server part of PFS. The server can choose to restrict this view by not exporting its local mount points. A different approach is taken whenever another PFS mount point is reached in the server file tree (see next section).

---

1. Paul Roy, from the Cambridge Research Institute.

Remote mount points

If a PFS mount point is created by the client on a directory which is already remote, the mount operation is forwarded to the server so that the mount point is created in its file space. Therefore this mount point will be visible by all clients accessing this part of the server file tree. At the same time, a cached copy of it is created on the client which has done the remote mount.

Transitive mount points

Whenever a client crosses another PFS mount point on the server tree (or when it creates a remote mount point), a local mount point is dynamically created in the client. This is achieved by creating a mount entry in the client mount table that is a cached copy of the server's original mount point. From then on, all subsequent client accesses that go beyond this point will be forwarded to the right server. The intermediate server only acted as a agent for the first access, by duplicating the send right it has for the remote directory, and sending it to the requesting client. Therefore, the mount point becomes transitive.

3.2 Mount point coherency

Coherency between cached copies and actual mount points is achieved by a revocation mechanism using Mach ports.

Mount table coherency

Whenever a client requests a copy of an existing mount point, the requesting message carries a send right for a port associated with the mount table entry containing the cached mount point. The server stores this right on a per mount table entry queue. All subsequent server modifications of this mount point (mount updates, unmounts) will generate a cached mount point revocation. Since NORMA ports are global, the server doesn't even need to know which clients are caching its mount point.

A client may choose to refuse mount point revocation due to the fact that he still holds references to files that are beneath the cached mount point. In that case, the operation started by the server (or by another client), is aborted and the EBUSY error is returned.

Name space coherency

There is also a mechanism for adjusting mount point arguments to the differences between client and server name spaces: client A may accesses server B mount point on a different point of its name space than server B accesses server C. Therefore, for mount information to be coherent (i.e: mount command issued with no arguments), mount point names stored in the mount table entries need to be adjusted.

For example, if client 0 mounts server 1_/cluster/export/usr directory on its local /usr, and server 1 mounts server 2_/usr/bin directory on its local /cluster/export/usr/bin, whenever client 0 crosses its /usr/bin directory, a transitive mount point will be created. Issuing a mount info request in node 0 will produce:

2_/usr/bin on /usr/bin

while on node 1 it will produce:

2_/usr/bin /cluster/export/usr/bin.

This feature is used to enforce single system image.

3.3 Unmounting PFS

Any client or server having access to a PFS mount point can perform an unmount or mount update action.

Local mountpoints

If the mount point is local, all the outstanding cached copies of this mount are revoked, by walking through the revoke queue. If these revocations succeed, the unmount action is performed in a standard way.

Remote mountpoints

If the mount point is remote (or has become transitive) the unmount or update action is forwarded to the server handling the covered directory. If this entry point has been cached by other clients, revocations are issued to invalidate existing copies. If they succeed, the mount point is effectively destroyed.

Loop detection

In the current version of PFS there is nothing to prevent looping mounts to happen. It's up to the configuration settings to avoid such situations.

4. Extended lookup operation

As stated before, the standard *vnode* lookup operation has been extended to allow pathname forwarding upon PFS mount points traversal. In fact, the PFS lookup operation has become a complex function that implements multiple aspects of the PFS semantics.

4.1 Pathname forwarding

When the generic OSF/1 *namei* function detects a mountpoint, it calls upon the VFS layer to get the root *vnode* for the mounted file system, which it then uses to replace the covered *vnode* on subsequent pathname translations.

Whenever a vop_lookup translates into a *pfs_lookup*, this function doesn't return after the first component after the mount point is looked up. Instead, it ships the whole remaining pathname to the server which handles the remote *vnode*. Typically, a lookup message carries two send rights for remote *vnodes*: one for the remote "root" (actually the remote directory which has been mounted on), and another for the directory on which to start searching. The return message carries a send right for the *vnode* representing the final path if the translation succeeded.

In case the pathname has not been completely translated for some reason, the return message carries back an offset that indicates how much of it has been consumed. The namei pathname field is adjusted so that the standard lookup algorithm may proceed.

To implement this extended lookup operation, minimal changes (roughly 60 lines of code) have been applied to the standard OSF/1 namei function.

4.2 Remote mount point detection

Whenever the server part of *pfs_lookup* detects a remote mount point it stops translation and returns a special flag to the client, together with a send right for the remote covered *vnode*.

Then, before returning to namei, the client part of *pfs_lookup* issues a *getmount* message to

the server, using to the covered *vnode* send right. This has the effect of returning information about the remote point being crossed, as well as a duplicated send right for the "root" *vnode* on the final node. *Pfs_lookup* then creates a transitive mount point as described above, builds all the standard chainings between covered *vnode* and mount table entry, and returns to namei with the rest of the unresolved pathname.

At this point, the standard OSF/1 namei routine detects a mount point, and issues a VFS_-ROOT call to get the root *vnode* for the newly created mount point. From then on, subsequent lookup operations will be directly handled to the right server.

4.3 File Attribute Caching

When successful, the lookup message also returns attributes for the looked up file. Those attributes are stored on a *vattr* structure that is part of the file system dependent part of the PFS *vnode*. A revocation mechanism is used to keep these cached attributes consistent with the state of the file on the server (see next section).

The lookup message also carries a revocation token (send right) for the attribute cache entry that will be used in case of success, thus avoiding an extra message for sending the token to the server handling the file. All in all, this allows *stating* a remote file in one message in the worst case.

5. Revocation Mechanism

The revocation mechanism is independent of the objects being revoked, therefore it may be shared between mount point and file attribute caches. It is based on Mach port send rights: whenever a object is cached on a client, a port is created and named with its address. A receive and send rights are then inserted. The receive right is moved to one of the global port sets on which the client pool of threads are listening. The send right is handled to the server owning the object that is being cached, which means that, for future revocation actions, the client part os PFS becomes the server, while the server part becomes the client.

The server managing the object stores the send right on a per object queue. On the attribute case, this queue is appended to the *vnode* structure. On the mount point case, since we only cache PFS mount points, it is part of the file system dependent part of the PFS mount structure.

5.1 Attribute Revocation

Since a client may store attributes for any type of server file system, attribute revocation has to be initiated above the *vnode* layer on the server system. Achieving this without heavily modifying OSF/1 original code could easily be performed by using a stacked *vnode* approach. Although this mechanism does not exist in the current OSF/1 version, it will probably be integrated in future designs. In the mean time, we used a very simple approach which consisted of adding to the *vnode* structure a pointer to an additional vector of *vnode* operations, called *revoke_operations*, and changing the macros that define the standard *vnode_operations* to test for the existence of this vector. In case this vector is non-null, indirect calls are performed through it rather that through the standard one. This has the effect of doing the revoke operation, which on success call the standard *vnode_ops.*

The *revoke_operations* vector is set for a *vnode* whenever it has been accessed by a remote client on behalf of a lookup operation, and a cached version of its attributes exist. We can say,

using DFS terminology, that the *vnode* has been converted.

The *revoke_operations* and *vnode_operations* vectors only differ for those operations that modify the file attributes (such as setattr, delete, link, unlink, mkdir, etc...). Two types of revocations (delete and update) may be issued, according to the kind of action being performed on the file.

Attribute tokens are inactivated by an explicit request from the client, whenever the client discards its cached version of the attributes (*vnode* recycling, cache invalidation). This avoids the server calling back for revocations on those clients that are doing modifications on files.

5.2 Mount Point Revocation

This mechanism has already been described in the previous section. When changes are done by a client holding a cached mount point, the message forwarding the action to the server holding the mount point carries a send right for the mount table being modified (or deleted). This allows the server to identify the client doing the modification and avoid sending a revocation, since the client may have momentaneously disabled operations on his mount table element. This way, possible deadlock situations are avoided.

6. Remote Representation of Files

Whenever a client accesses a remote file by looking it up, it is given a handle by the server managing the file system on which the file lies. This handle is created by the server part of the *pfs_lookup* function, and consists of a send right to a port representing the server *vnode* (actually, the port is renamed with the address of the *vnode*).

The client stores this send right on the file system depended part of the *vnode* (we called it the *remote_vnode* structure) representing the remote file. Since NORMA ports rights implicitly designate the node number, no other location information is stored in this structure. This is an important feature for future process migration support.

Send right reference counting is also kept on the *remote_vnode* structure, so that the total number of rights is freed when the client *vnode* is recycled.

One extra user reference count is kept on the server *vnode* each time a client has remote references to it. Therefore, PFS is a statefull file system. This reference is freed by a no-more-senders message[1] that is sent to the server whenever the number of outstanding send rights to the *vnode* port drops to zero.

The server part *vnode* also keeps count of the number of send rights created to the *vnode* port as well as the sequence order of delivered messages, so that no outstanding messages are lost when a no-more-senders message is received. Part of these mechanisms have been adapted from ADFS.

7. Remote Special File Access

PFS allows access to remote special files through the *vnode* layer, and enforces no special naming conventions or node identifiers in the special file representation on disk. This way, remote files on one node are simply accessed by other nodes by explicitly doing:

---

1. This is subject to change, since using no-more-senders for this purpose is a bit too expensive and a simpler solution may be implemented.

mount <node #>_/dev on <localdir>

and issuing normal requests as if the files were local. The operations on special files are thus forwarded to the remote node, performed locally, and the results are returned back to the client node through Mach IPC.

Although this may change in the future and we may be forced to use Mach IPC to directly access files from remote nodes (as it is done in AD), we chose to implement this solution in the first place because we think it is the one that is best adapted to process migration.

In fact, it allows all state associated with devices to remain in the server handling them, since the file operations are effectively performed by this server and not the client doing requests, as it would be that case if Mach IPC would be used at the device driver level. In case of process migration, only the part above the *vnode* layer needs to be migrated, and the part underneath it, that typically contains most of the state associated with special files, will remain in the server handling the device.

Of course this solution have other drawbacks. Not having a way of specifically naming remote devices precludes the possibility of having nodes with no servers. On the other hand, the number of data copies needed in the present case is not minimal. Perhaps both types of accesses (at *vnode* and device driver levels) should be allowed in the future.

We need also mentioning the fact that in the current implementation of PFS the select and ioctl *vnode* operations are not implemented, so remote special file access has only been validated for block devices: fsck on remote partitions works.

8. Single System Image

Due to cluster initialization issues, PFS does not allow an automatic view of all the disk resources of the participating nodes under a single system image. This is mainly related to the fact that the current cluster configuration is still an open issue, and that some important work needs to be achieved in order to develop a cluster configuration server.

However, PFS is able to offer a single system image on a partial subtree of the cluster file system that we decide to share and manually insert under PFS mount points. Figure 1 gives a simple example on how a cluster of 3 nodes can provide single system image for the /usr directory. Curved arrows represent PFS mount points, while straight ones represent local mount points or hierarchical directory links.

Suppose we want to distribute the contents of the /usr directory between 3 nodes in the following way:

1) node 0 has the /usr directory;

2) node 1 has /usr/bin directory;

3) node 2 has /usr/bin/X11 directory.

Node 0 does:

- mount 1_/bin /usr/bin.

Node 1 does:

- mount 2_/X11 /usr/bin/X11
- mount 0_/usr /usr

Node 2 does:

- mount 0_/usr /usr (or mount 1_/usr /usr, like in Fig. 1)

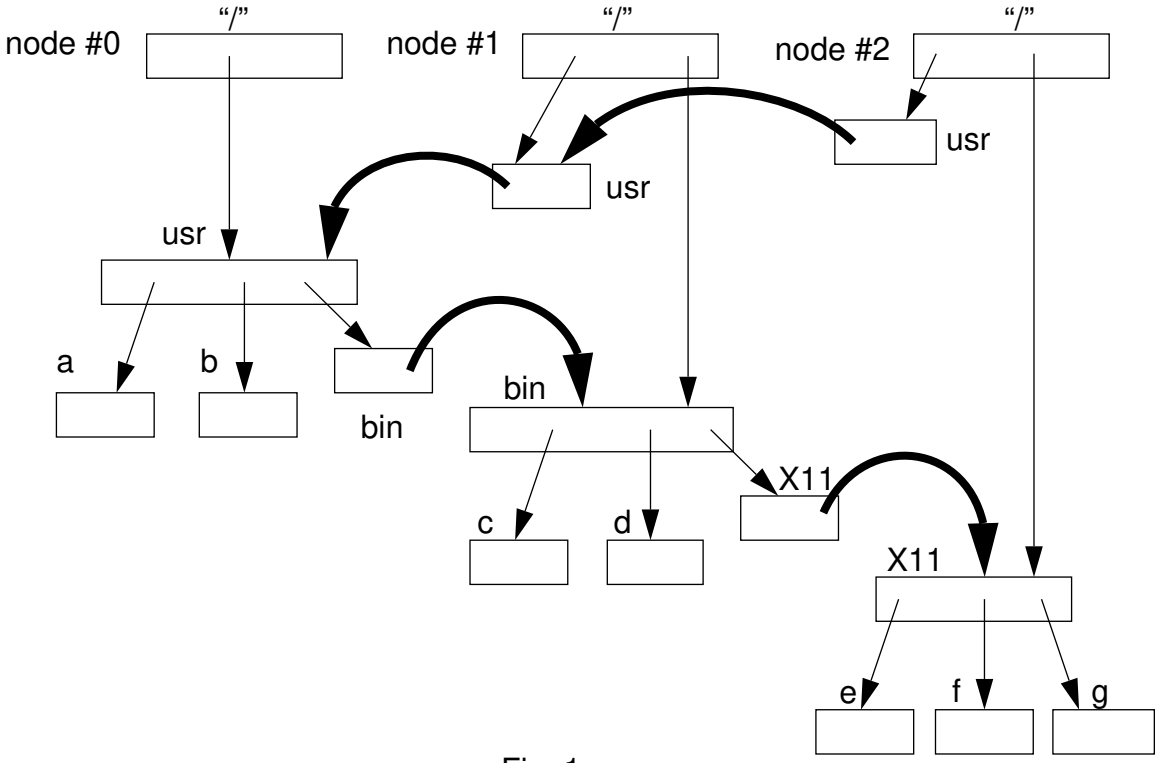

Fig. 1

From now on, any other node mounting the /usr directory of any of the three previous nodes will have the same exact view. Even if other subtrees are dynamically added or removed from any /usr subdirectory located on any node.

Of course, what we really aim at is to be able replace "/usr" by "/" in the example above, so that all nodes share the exact same view in all of their file space.

We can also see in this example that in some cases some directories are visible through two different paths: that's the case, for example, of node 2 and directory X11 which is visible through PFS as /usr/bin/X11, and as /X11 through local access. This will also not be the case when the local root will also be visible through PFS. Of course, a local access will be optimized so that no messages are issued when local resources are accessed through PFS.

9. Process Migration Support

PFS does not currently support process migration. But the fact that its architecture is based on the *vnode* layer simplifies moving state from node to node when a process with open files is migrated. Some preliminary design considerations can be done at that time.

Accessing local files through PFS

In order to allow file information mobility, only processes accessing PFS files on both source and destination nodes are allowed to migrate, which implies that all exported local file resources should be accessed through PFS, which is achieved by mounting all the exported local partitions under PFS. This in turn forces the loopback path to be optimized, so that local accesses are performance neutral with respect to direct accesses.

An optimized loopback path can be implemented by introducing a virtual layer at the stub level, so that either MiG stub routines or local wrapper functions are called, depending on the locality of the handle port. The vector for the stub layer will be located in the *remote_vnode* structure and will be initialized when the *vnode* for the file is allocated.

On the server side, provision has also to be made so that reference and send right counting is done differently when the server routines are called locally.

Migrating PFS Vnodes

In the simplest case, when a process with an open file migrates, the state above the *vnode* layer and the PFS related state needs to move. If the process shares access to the file with other unrelated processes, this state will be duplicated. This state includes the *vnode* (both independent and dependent parts), the file structure, locks, etc... On the destination node, place to accommodate this state has to be created, so the file needs to be reopened. But instead of being initialized with default values, those structures will receive their initial state from the source node. Mainly, the port file handle that is part of the PFS *remote_vnode* structure will be migrated and no other location dependent action needs to be performed.

No state related to file system dependent information needs to be transferred, since it will stay in the server actually managing the file. Only the PFS *vnode* has been migrated to allow access from another location.

Processes with shared file structures

A more complicated case happens when processes sharing file structure with its ancestors are elected for migration. A first drastic solution is to prevent such processes to migrate. Although this is perhaps not a very important restriction, a solution can be found that allows such processes to move.

Since the file structure lies above the *vnode* layer, allowing concurrent accesses from unrelated local and remote *vnodes* to the file does not allow keeping the file offset coherent. In this case, local and remote file structures have to be accessed under a mutual exclusion protocol that may be implemented using a token mechanism.

Such a mechanism will also be needed for other file system operations, such as data cache coherency. We therefore plan to either implement a general token mechanism in PFS, or to adapt any existing available one (see next section).

10. Integrating new technology

The OSF Research Institute has recently issued a Request for Technology to be integrated in the next version of OSF/1 for Multicomputers (OSF/1 AD), so that this system will simultaneously be able to manage a cluster of networked workstations.

On the file system area, there are requests for high availability and scalability, as well as load sharing through the use of multiple loosely coupled disk nodes. On the other hand, one explicit request is made about the layering of such functionality and their integration on the existing OSF/1 *vnode* and VFS interfaces.

Besides, the integration of new non-OSF technology in the current OFS/1 AD system will need a partial redesign of its low level functionalities in order to cope with the underlying OSF/1 AD paradigms, which are NORMA Mach IPC and XMM.

As we have just described, PFS combines the use of such NORMA Mach technology with existing OSF/1 interfaces to achieve the basic functions of a distributed file system.

On the other hand, PFS implementation has been done in a standard no-emulator OSF/1 server with a minimal impact on the existing code.

Therefore, we think that PFS can be a suitable platform on which the new acquired file system technology can be inserted. In order to achieve that, we propose to finalize the existing PFS prototype with the following extensions.

11. Completing PFS prototype

As stated across this document, the following PFS functionalities are incomplete or are not implemented at all.

Pathname cache

As mentioned before, there is a design for a multiple component pathname cache, that relies on part of the existing VFS lookup cache. We think that this is the main missing piece of fuctionality, and therefore its implementation should be done immediately, although some extra design work has to be done to determine a way for correctly invalidating cached entries.

Mapped file access

In the current PFS implementation there is no data cache on the client part. The reason for leaving such an important issue unresolved is that we plan to integrate the mapped file module from OSF/1 AD under the *vnode* layer. This will probably be done in very close collaboration with the Cambridge RI. Besides, a token management scheme for data coherency also needs to be designed.

Loopback optimization

Although this is not mandatory for functional completeness, no performance measurements can be done without avoiding messages when local files are accessed. A design for this mechanism has already been described in a previous section.

Remote special file access

Access to remote character special files trough the *vnode* interface still needs the *select* and *ioctl* operations to be implemented.

Remote special file naming

A way to access nodes with no servers need to be provided. The current *vnode* access supposes remote operations on files to be done by a local server. The OSF/1 AD mechanism to name remote devices by inserting the node number in the disk representation can be adopted. We think that a global design decision has to be taken in that subject.

Single System Image starting from root

Once all types of devices are accessed through the PFS layer and the loopback path is optimized, the local root may be mounted on PFS. The remaining problem has to do with taking decisions on how the file system is distributed across the cluster disk nodes. This problem is very closely related with cluster configuration. As there is a proposal for work on that area[1], we will not cover this issue for now.

---

1. See document by Eric Paire

12. Conclusion

In this document we described PFS, a prototype distributed file system based on NORMA Mach IPC and the *vnode* interface. Although it is functionally incomplete, we believe that quite a chunk of the total work has been achieved, and that the existing prototype is a demonstration of feasibility of its main announced features.

To make PFS evolve to a real usable file system still implies a reasonable amount of work, that can only be achieved in close cooperation with the Cambridge Research Institute. On the other hand, the result of this work will be a scalable file system basis that can either be used alone, or extended on demand with third party functionalities to achieve more sophisticated distributed environments.