The InterMezzo File System

Peter J. Braam, *braam@cs.cmu.edu*
Carnegie Mellon University & Stelias Computing
Michael Callahan, *mjc@rodagroup.com*
The Roda Group
Phil Schwan, *pschwan@inter-mezzo.org*
Stelias Computing

Abstract

Is it possible to implement a distributed file system, retaining some of the advanced protocol features of systems like Coda, while getting an implementation that is an order of magnitude simpler? We claim the answer is "yes", and InterMezzo is well on its way towards being a prototype of such a system. The key design decisions were to exploit local file systems as server storage and as a client cache and make the kernel file system driver a wrapper around a local file system. We use a rich, high level language and work with asynchronous completion instead of threads. Finally, we rely on existing infrastructure, such as TCP. This is in contrast with systems like Coda and AFS, which implement their entire infrastructure from the ground up, in C. This paper will describe the design and implementation decisions used in our prototype.

Introduction

Designing and implementing distributed file systems poses a major intellectual challenge. Many excellent ideas have come out of research projects, and some systems offer many advanced features.

The basic question we wanted to answer is the feasibility of building an interesting distributed file system with an order of magnitude less code than a system like Coda [coda] has, yet retaining some of the advanced features. We claim the answer is "yes". In this paper we will describe some of the design and implementation decisions for the InterMezzo file system.

To keep our project modest, we are first targeting the project of online directory replication on a system area network, allowing for network and system failures. This exposes many of the subsystems needed, but leaves some others aside. However, our paper describes the broader design issues, parts of which will be implemented and likely modified in the process of bringing InterMezzo forward.

The code for InterMezzo will soon be available at http://www.inter-mezzo.org.

Acknowledgements: The authors thank the other members of the Coda project for fruitful discussions. This research was supported by the Air Force Materiel Command (AFMC) under DARPA contract number F19628-96-C-0061. IBM and Novell provided additional support. The views and conclusions contained in here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, DARPA, IBM, Novell, Carnegie Mellon University, Stelias Computing Inc., The Roda Group LLC or the U.S. Government.

Designing Distributed File Systems

There are several components to distributed file systems. First, network file systems have *clients*, which can fetch data from *servers* and send modifications to servers. This requires network request processing including bulk transfer. The fact that the client gains file system access to the files requires a kernel level file system driver. Without adding bells and whistles here we already have a list of complicated components.

Delicate protocols and versioning information is needed to give good semantics of sharing. For example, a server may have to notify clients that cached versions of the files or directories are out of date. This requires the client to both make requests to servers, on behalf of client processes accessing and/or modifying file system objects, as well as to answer requests from the server to maintain cache validity.

Performance is another important consideration, which usually becomes pervasive in a design. One must minimize the number of network interactions, and also make these interactions as efficient as possible. Security, management and backup add another dimension to a file system project.

Given this picture, it should not come as a surprise that not all the advanced file system projects have delivered systems that gained widespread use. Fortunately many features and ideas have been explored and thoroughly evaluated during the last 15 years. A central question that has not been answered appropriately is how to implement file systems reliably, with a reasonable amount of code.

Here something can be learned from other distributed systems. Ericsson was facing the daunting task of implementing control systems for telephone exchanges. The Erlang language [erlang] was developed as a result of extensive studies how to construct reliable distributed systems. The Teapot [teapot] project at the University of Wisconsin explored domain specific languages enabling construction and verification of distributed shared memory software. This toolkit was subsequently used by the Berkeley XFS project [xfs]. The ACE toolkit [ace], developed at George Washington University, provides extensive C++ frameworks for building distributed systems.

Interestingly all of these projects have reached conclusions bearing considerable similarity: a state machine with asynchronous event notifications is exploited, which gives a better overview of the distributed state than the classical "multi-threaded" server models.

Another key contributor to difficulties with file systems lies in the kernel code. Linux development has shown that even to get a correct implementation of a relatively simple file system such as NFS, numerous subtle issues have to be addressed, resulting in a long arduous development path. UCLA's Ficus [ficus] project made an important contribution in this field, namely to use stackable file systems [stackable], also available for Linux [zadok]. These are file system **layers** that leverage and extend the functions of other file systems. We use this approach in InterMezzo and avoid the difficulties associated with a full file system implementation.

InterMezzo is at present a prototype, and the first target for the system is to do reliable directory replication between two nodes, on a secure system area network. Its protocols are suitable for extensions, and a fully featured distributed file system can be constructed from this starting point.

Overview of InterMezzo

Faced with a multitude of design choices, we drew up the following list of requirements.

1. The server file storage must reside in a native file system.
2. InterMezzo's client kernel level file system should exploit existing file systems, and have a persistent cache.
3. File system objects should have meta-data suitable for disconnected operation.
4. Scalability and recovery of the distributed state should leverage scalability and recovery of the local file systems.
5. The system should perform kernel level write back caching.
6. The system should use TCP and be designed to exploit existing advanced protocols such as rsync for synchronization and ssl/ssh for security.
7. Management of the client cache and server file systems should differ in policy, but use the same mechanisms.

Many issues remain open. For example, we have not yet decided if we should identify InterMezzo file objects (i.e. files & directories) through file identifiers, path names or

server/device/inode triples. Our system is designed to enable us to switch from one model to another with a relatively small effort.

Security and management are important. AFS and Coda set good examples for management of server space, by working with file sets, a.k.a. volumes. File sets are groups of files that form a sub-tree of the InterMezzo name space, and which are typically much bigger than a single directory and much smaller than an entire disk partition. Coda and AFS, as well as DCE/DFS and Microsoft dfs, have a *single name space*. This means that all client systems see one large file tree containing all the file sets exported by InterMezzo servers in a cluster.

Often it is desirable to work with a small collection of file sets separately, so we left room to manipulate the InterMezzo exported file sets more explicitly than in those systems.

File sets, mount points, servers and clients

Each file tree made available by a cluster of InterMezzo servers is built up of file sets or volumes, which are similar to Coda and AFS volumes. Each file set has root directory and can contain InterMezzo mount points of other file sets. An InterMezzo mount point is a concept similar to but distinct from a Unix mount point. A client can have any file set as the root of an InterMezzo file system.

A file set has a file set storage group associated with it, describing the server holding the file set. The file set location database describes the mount points of file sets, as well as their storage group. An InterMezzo cluster is a group of servers sharing a single file set location database (FSLDB).

A client will be able to choose any file set as the root for a mounted instance of an InterMezzo file system. If the root file set is not the default, then all ancestors of the mount point of that file set are invisible, this accommodates exporting sub-trees of the entire cluster file tree. File sets contain files and directories, and generally user level software is not aware of the presence of file sets.

The file set location database is implemented as a sparse tree of directories, handled as a special object in the file system. An update record is associated with each modification of the file set tree, and version numbers are associated with each update. This allows the file set database to be updated, without downloading the latest version (which is also kept).

InterMezzo makes a distinction between clients and servers, but mostly as a matter of policy, not mechanism. While the same mechanisms are used, the server maintains the on-disk structures as authoritative records of file state, and also coordinates state coherence across the InterMezzo clients. Each server will also be an InterMezzo client, but the file system layout on the server is slightly different from that on other clients. A file set mount point on a server is not necessarily a directory in the InterMezzo file system as it is on a client. If the server stores the file set, a symbolic link is placed at the mount point, which targets a directory on the server holding the data. This target directory is the file set holding location (FSHL).

Journaling updates and filtering access

An InterMezzo cache is simply a local media file system, which is mounted with an extra filter layer wrapped around this local file system. The filter layer is called "Presto" for Linux and "Vivace" for Windows platforms. The purpose of the filter is twofold:

- Filter access, to validate currency of the data held in the local file system
- Journal updates made to the file system

All requests emanating from Presto are handled by Lento, the user level cache manager on the system. Note that Lento acts both as the file server and as the client cache manager. Figure 1 graphically describes the operation of the InterMezzo filter driver.

The code in Presto is quite straightforward. When InterMezzo is mounted, Presto is

informed of the file system type that it is wrapping. It remembers all VFS methods associated with dentries, inodes and files of the wrapped file system in the "bottom_ops" structures. The "bottom_ops" are typically those of the ext2 file system. Presto must filter all access and journal all updates, but must make an exception for the process managing the cache, which is named Lento and will be discussed later. Lento's modifications of the cache should be immune from filtering or journaling.

As an example let us describe the open call for directory inodes. When an object is accessed and it is not present in the cache, the object is fetched. Files are presently fetched in their entirety. Directory fetching is implemented by creating the name, and creating sparse files and empty subdirectories for each of their entries. While this introduces significant latency in the access to directories, compensation comes from the fact that the subsequent acquisition of attributes of the entries is entirely local.

In pseudo code, the kernel open method has the following structure:

```
int dir_open(inode *inode, file *file)
{
        if (!HAVE_DATA(inode) && !ISLENTO)
        {
                upcall(inode, "get_data");
        }
        return bottom->dir_open(inode, file);
}
```

Updates to objects are made as follows. First a *Permit* is acquired to execute concurrency control. On clients, the updates are made in the cache and journaled in a client modification log for *reintegration* to servers. The server holding the file set repeats the modifications made to the volume and will forward the modification log to clients registered for replication. We call the process of forwarding modification logs to the clients *update notification*. The processes of reintegration and update notification are almost identical.

The modification logs details modifications made to directories and file attributes, as well as all close operations on files. Such files, which have been closed after a modification, are then "back-fetched" by the server. If a server detects that it does not have a directory created by the client, it recursively back-fetches the contents of that directory from the client.

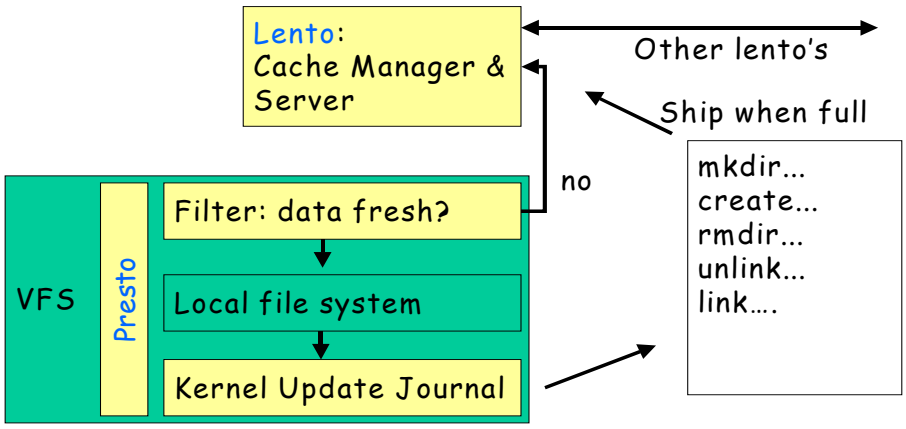

Figure 1: Basic operation of the InterMezzo system.

The mkdir method in Presto is roughly implemented as follows:

```
int mkdir(inode *inode, char *name)
{
        if (!HAVE_PERMIT(inode) && !ISLENTO)
        {
                lento_getpermit(inode);
        }
        rc = bottom->mkdir(inode, name);
        if (! rc  && ! ISLENTO)
        {
                journal("mkdir", inode, name);
        }
        return rc;
}
```

A client can place a replication request for a file set with the server holding the file set – the server will then propagate all updates it receives to this client.

If a file set is modified on the server, an almost identical process takes place. The updates are journaled but now only propagated to clients registered for replication. Other clients will invalidate their caches when they learn of the updates, and re-fetch the data. In order to accomplish this, the holding location of a file set is itself mounted as an InterMezzo file system to enable journaling of updates.

We see a high degree of symmetry in the operation of clients and servers, and Lento running on clients and servers must be capable of:

- File Service: fetching (on clients) and back-fetching (on servers) of files and directories
- Reintegration service: receiving modification logs (on servers) and update notifications (on clients) for reintegration

We see that both servers and clients should be capable of making modifications and serving data.

A key distinction is that a server implements security while a client can trust an authenticated server and implement changes without checking permissions on an object basis. Secondly a client may make changes to its cache solely for freeing up space. Such modifications are, of course, not propagated to servers.

In a trusted environment, an optimization can take place, which simply changes the holder of the file set to the system modifying the file set. This system then is responsible for updating all replicators of the volume.

Protocols

The Coda, AFS and DCE/DFS protocols (see [dfs-protocols] have many attractive features, and our intention was to preserve many aspects of these.

Each cached object (i.e. directory or file) has attributes HAS_DATA, HAS_ATTR to indicate if its data/attributes are valid in the cache. Before accessing an object, the client will verify that it is current, and re-fetch it if necessary. The protocol has a **FetchDir** and **FetchFile** call for fetching and a **Validate** call for assessing currency. The presence of a HAS_DATA or HAS_ATTR flag is called a call-back and enables the client to reuse the object without contacting a server.

The server will notify clients holding callbacks on objects before storing modified objects. InterMezzo's design could also allow breaking these callbacks before the modification is initiated. In either case, a server-initiated request **BreakCallback** is issued. The BreakCallback request is handled as a multi RPC, as in Coda. This means that the BreakCallback request is sent out to all clients and only then replies are gathered -- this avoids multiple timeouts. When data is to be modified, a client will get a permit to do so. There is a **GetPermit** request to do this. A successful permit acquisition is indicated with a HAS_PERMIT flag in the status field.

Every object will be identified by an identifier and a version stamp. Our protocols will guarantee that two objects with the same identifier and version-stamp are identical. In addition to object version stamps, volumes have stamps too, and allow for rapid revalidation of entire volumes, in the common case where no objects in the volume have changed.

InterMezzo can validate cached objects with **Validate** requests. As in Coda, these can be

issued at the volume and file system object level.

The propagation of updates is done through the **Reintegrate** request. This request uses the *modification log*, in a way that is similar to Coda's reintegration. The client modification log (CML) is shipped to the server. The server proceeds to incorporate the changes to the directory tree and then fetches the files from the client, for which close operations were in the CML.

Coda avoids many race conditions by including the version with the RPC for the file affected by the update. For example, the creation of a new directory would include the version of the parent directory present on the client. This allows the server to make sure that the server version being modified is that of the client. If the versions are unequal, the reintegration of this modification causes a conflict, which needs special handling. InterMezzo will do this as well.

So at present InterMezzo's protocol is very simple.

Lento – cache manager and file server

Lento is responsible for handling file service requests from the network or the kernel. Traditional servers and cache managers are implemented using a thread pool model. The requests come in and are placed on a queue. Worker threads pick up and process the requests, and block while doing I/O, during which other worker threads can continue.

We chose to do start with a single threaded event driven implementation based on asynchronous I/O. In Lento, requests come in and a session is instantiated to handle them. A *session* is not a thread, but a data structure that contains state and event handlers. I/O is done asynchronously and a *kernel* signals the completion of I/O, through dispatch of events. The kernel activates all event handlers and is also responsible for garbage collecting sessions when no events can reach them. Sessions have process style relationships.

For prototyping we used the POE (Perl Object Environment) toolkit, see [poe]. POE implements a session framework as described above and has *Wheels* that consist of *Drivers* to do the I/O and *Filters* to pre-process the data received.

We added two wheels to POE. The PacketWheel is there to send and receive network requests, which the filter unpacks and then gives to the connection. The connection determines the destination session for the request, which can be an existing session or the request dispatcher, in case a new or specially chosen session must handle the request. Our UpcallWheel unpacks requests originating in the kernel, which reach Lento through a character device /dev/presto.

These wheels have been combined with an AcceptorWheel that creates accepted TCP connections.

The processing of net and kernel requests is graphically indicated in figures 2 and 3.

As an example we will give the pseudo code for the upcall session servicing an "FetchFile" request from the kernel. The notation below describes a session as a hash of event handlers.

```
Fetchfile = new session (
{ init => {
        if (!have_attr)
                req_attr(attr_arrived);
        else
                post(fetch_data);
        },
attr_arrived => {
        if (status == success)
                post(fetch_data)
        else { destruct_session(error);
        },
new_filefetch => {
        queue_event(this) ;
        },
complete => {
        reply_to_upcall;
        handle_queue;
        destruct_session;
        }, ……
} );
```

Each of the event handlers may engage in a blocking I/O operation, for example to fetch the

attributes. In doing so it indicates what event will complete the asynchronous operation. Ultimately, when the data is all on the client, the request completes to the kernel by replying to the upcall. A queue associated with the session allows further requests to fetch the data for the same object to be handled by the session that is already in the process of fetching.

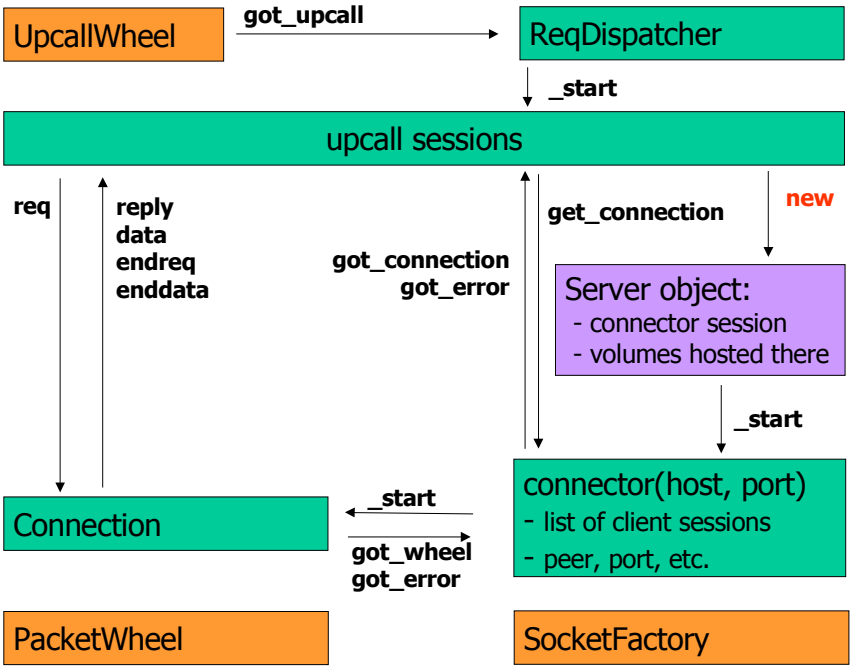

Figure 2: Upcall Handling

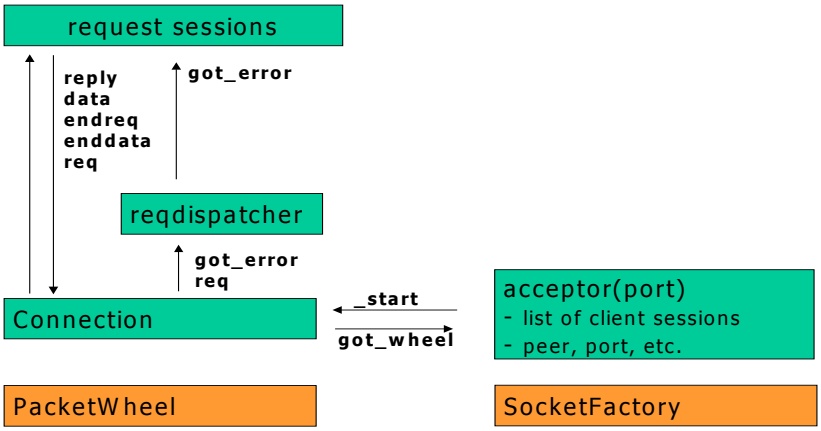

Figure 3: Network Request handling

Lento also needs a variety of daemon sessions, which are not garbage collected but perform simple periodic or persistent tasks. For example, we have the ReqDispatcher, which sets up new request handling sessions. For the management of client initiated connections, we

use Connector daemons, which create a TCP socket connect to a server, and manage the sessions using the connection.

Network packets contain Asynchronous Completion Tokens (ACT's, see [ace]) which encapsulate the source and destination sessions for request handling. There are 6 types of packet: *requests, messages, replies, data, end-of-data and end-of-request packets.*

Lento also uses various tables: tables exist for servers, volumes, and file system objects, each with their own attributes. When Lento starts a number of data structures are set up, as depicted in figure 4.

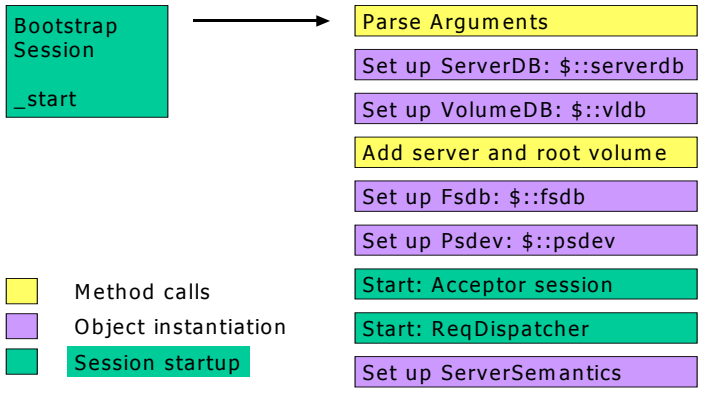

Figure 4: Lento startup

Some of the set-up processes are quite involved. For example, setting up the file system database (Fsdb) may involve recovery. (Currently this consists simply of wiping the cache.) The server semantics supplement the Fsdb database with callback and permit tracking, as these are released and acquired by clients.

Recovery and cache validation in more depth

On Linux, InterMezzo can use the ext2 file system to hold server and cached data. We envisage having a fast and sophisticated file system database (Fsdb) for extra meta-data required by InterMezzo. This would be managed by Lento, and would hold the version information for cached objects, as well as inode-to-filename or inode-to file identifier translation information. If a client crashes some of the data will be held in buffers, other data will be on the disk and it is necessary to think through precisely what is needed for clean recovery.

The simplest recovery process is to simply wipe the cache, which for large caches is clearly very undesirable. The basic attribute for establishing currency of a cached object is the version stamp. If the version stamps of the objects are equal the objects should be equal, and version changes originating from a single client should be ordered.

Consider the situation of a crash while fetching data into the cache. The situation to avoid is to believe that the cache is up to date, while in fact the data was not fully fetched. To achieve this, the version stamp should not be written to persistent storage until it is certain that the fetched data has made it to the disk. On Linux this can be achieved by simply waiting 30 seconds before flushing the Fsdb database buffers. Upon reconnection to a server, all objects having an out of date version stamp should be flushed from the cache. If recovery is followed by disconnected operation, one could allow the possibly incompletely fetched objects to be used, for maximum availability. Such objects can be identified as those having a ctime less than 30 seconds since the last Fsdb flush. If such objects are modified and reintegrate, they be regarded as "suspect" and be handled as though they were conflicts.

If a client is modifying existing data when crashing, then the reverse risk exists: the client,

when recovering, may see an old version stamp in the database, and possibly also an old mtime in the inode. If the object has just been fetched, the version stamp may even be older than that on the server. Yet the data on the client disk may be newer than that on the server. There are several approaches here, the simplest being "laissez faire" and accepting the fact that a few seconds of data modifications might be lost. This is not very desirable, since everything should be done not to lose the latest version of data.

Alternatively, the inode under modification could be updated with a new mtime and synced to disk. Provided modifications do not happen within the granularity of mtime units this allows the recovery mechanism to identify this file as suspect. The collection of suspect files is defined as those having an mtime later than the last sync time of the Fsdb. Such suspect files could be made subject to rsync style reintegration.

Yet another solution is to write the Fsdb database record to disk, before starting the modification, indicating that the inode has now changed. The latter involves a context switch to Lento, but since disk traffic is involved anyway, this is probably not a significant overhead, see [bottlenecks].

This detailed and slow initiation of modification procedure would not affect newly created objects.

InterMezzo aims to be usable for daemon controlled situations, for example to provide replication for WWW data. In such cases handling conflicting updates should not involve user interaction and we will introduce policies to handle conflicts automatically, for example by giving preference to the server copy and notifying an administrator of a conflict.

InterMezzo on Windows 9x, Windows 2000 and other Unix systems

We believe that the InterMezzo system can fairly easily be ported to Windows 9x and Windows NT/2000, building on the experience we gained when doing this for Coda [coda-win].

Both systems have the capability to do sophisticated filtering of file system requests and combine them with up-calls. Simple examples of such filter drivers are found in the FileMon utility [filemon]. For Windows 9x an additional difficulty arises from the non-reentrancy of the operating system. Lento will have to be a 32-bit DOS application, just like Coda's cache manager Venus. Fortunately, the Coda project provides the tools to make a port possible, and somewhat amazingly the Coda DOS cache manager works very well.

Porting InterMezzo to other Unix systems should be possible, but might require knowledge only obtainable through kernel source.

Conclusion

With 2,500 lines of C kernel code and 6,000 lines of Perl code (don't worry, no one liners here) we have managed to implement the basic functionality of the InterMezzo file system.

This is very encouraging and we hope to continue towards a fully functional robust file system.

References

[ace]
http://siesta.cs.wustl.edu/~schmidt/ACE.html

[bottlenecks]
Removing Bottlenecks in Distributed Filesystems: Coda and Intermezzo as examples, Peter J. Braam & Philip Nelson, Linux Expo 99.

[coda]
http://www.coda.cs.cmu.edu

[coda-win]
http://www.usenix.org/events/usenix99/braam.html

[dfs-protocols]
Distributed File Systems for Clusters: a Protocol Perspective, Peter J. Braam, Usenix Technical Conference, Extreme Linux Track, 1999.

See also:
http://www.extremelinux.org/activities/usenix99/docs

[erlang]
http://www.erlang.org/

[ficus]
See: http://ficus-www.cs.ucla.edu/travler/

[filemon]
See: http://www.sysinternals.com

[poe]
See: http://www.netrus.net/users/troc/perl.html

[stackable]
John Shelby Heidemann. Stackable Design of File Systems. Technical Report CSD-950032, University of California, Los Angeles, September, 1995.

See also:
http://www.cs.columbia.edu/~ezk/research

[teapot]
Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI), May 1996.

[xfs]
See: http://now.cs.berkeley.edu/