Lean-DFS: A Distributed Filesystem for Resource Starved Clients

Shyam Antony and Gautam Barua

Dept. of CSE, IIT Guwahati
Guwahati 781039, India
gb@iitg.ernet.in, antony@cs.ucsb.edu

Abstract. Devices which have limited computing resources but are capable of networking have become increasingly important constituents of distributed environments. In order to maximize the advantages of their networking capability, it is important to provide efficient methods to access modify and share data. In this paper we position distributed filesystems as a possible solution to this problem. We discuss the constraints imposed by such environments on traditional distributed filesystem design parameters. We describe the design and implementation of a distributed filesystem (lean-dfs) for such environments.

1 Introduction

Devices which have limited resources in terms of computational power, memory etc. form vital components of 'intelligent' environments. Such resource starved machines are increasingly capable of networking. In order to maximize the benefits of networking, it is necessary to provide efficient methods to access, modify, and share data. A Personal Digital Assistant (PDA) cum cell phone needs to store information it gathers, in a server. Current PDAs have applications that "synchronise" files between the PDA versions and the server versions. A general purpose distributed file system will be a better vehicle for providing support for all these activities than piecemeal, machine and OS specific solutions for each of the different usages. Embedded systems need to communicate with "Master" nodes to get commands and parameters that change its behaviour from time to time. The use of configuration files to control the behaviour of processes is a well understood and time tested technique So a distributed file system could be the vehicle of communication in an embedded system network, rather than specific protocols for specific needs. Distributed Filesystems (DFS) form an integral part of distributed environments and have been adapted for a variety of research goals. General information on DFS can be found in [1, 2]. NFS [3–5] is the most widely used distributed filesystem. There has been some work on DFS for mobile clients, notably, CODA and Odyssey ([6, 7]). Recent work by Atkin and Birman and by Muthitacharoen et. al, have focussed on a low bandwidth environment ([8, 9]). Both works do not put constraints on resource availability other than bandwidth. Software for PDAs devote attention to the building of

file systems on flash memory with its specific characteristics and these have no direct relationship with a distributed file system. The PalmOS, a popular PDA OS, has facilities for file synchronisation and file copying as a result, but no distributed file system interface is provided ([10]).

2 Constraints and Implications

Memory Constraint: Resource starved clients usually have very limited main memory. This rules out implementation choices that involve high memory usage such as a large main memory file cache or a highly sophisticated protocol the implementation of which will involve a large code size.

Lack of Non-Volatile Storage: Since resource starved clients usually do not have access to significant local non-volatile storage, distributed filesystem design paradigms which implicitly convert a remote file into a local file, as in [8], are not possible.

CPU and Power Issues: Unlike normal clients which are designed to multiplex a number of applications, resource starved clients are designed to execute one or two applications at a time. As a result, no process can execute on behalf of the distributed filesystem in the background. For example it is not possible to execute a daemon which lazily updates a cache. Further, features such as callbacks, delayed writing and read-ahead, relying on background processes cannot be used.

Network Issues: Earlier work on mobile file systems have included a lack of bandwidth as a constraint. Even though network bandwidth is limited, it is not a constraint nowadays. So a tradeoff by using more network bandwidth to preserve other resources is feasible. This translates into the reduction of caching resulting in more network traffic.

Application Characteristics: Unlike normal clients, in a resource starved client, files are not manipulated directly by users but indirectly by applications. Hence the workload for distributed filesystems for resource starved clients is different from conventional file workloads.

TCP Vs UDP: This is not a constraint but more of a implementation choice. The recent trend among distributed filesystem implementations is to use TCP. However since UDP is more lightweight than TCP, our implementation uses rpc over UDP.

3 Lean-dfs Filesystem

Typical Workloads In the case of resource starved clients, we have identified two most common workloads: *Workload I:* This workload consists of very short reads and writes with long periods of inactivity between successive operations. Typical examples include reading configuration and startup files from the remote server and writing to remote files to indicate some change in the client's environment. The long interval between operations occurs because changes in

the environment happens only occasionally. In order to be useful it is necessary to provide rapid response time to such reads and writes. *Workload II:* This is the large file workload which usually consists of large files which are read or written sequentially in fairly large chunks. A typical example of this workload is the processing of multimedia files.

3.1 Optimisations

In the following paragraphs we outline a few optimizations for workload I to provide a rapid response time. Conventional features of NFS are sufficient for workload II since such a workload cannot benefit from caching due to sequential flooding.

Mounting: Since resource starved clients often suffer long periods of inactivity between periods of activity, it is necessary to allow a rapid 'mount-operate-umount' paradigm. The client sends an rpc request called MOUNT to the server. In response the server returns the root filehandle and server side parameters such as maximum allowed file name etc. No security checking is done at this point and hence any client can execute the mount request. In order to reduce the time taken by the server to browse the server exports, we insist that the server export a single, unified filesystem tree. As a result of these changes we were able to achieve rapid mount times of around 100 ms with a server under moderate load.

Truncate-on-Write and Consistency Issues: When a write is intended to over-write a file completely and the length of the new write is shorter than the original file size, a write request carries an extra variable called truncate which as suggested by its name, truncates the file before writing. The consistency provided by this scheme is extremely weak but quite sufficient for workload I.

Limited Batching with Compound Requests: Since the constraints of resource starved clients force the distributed filesystem to be an 'on-demand' system, we can ensure a rapid response time by batching requests which are related into a single request. This is similar to the batching in nfs version 4 ([5]) and the 'andx' feature in cifs ([11]). However since we use udp rather than tcp, arbitrary batching of requests cannot be accommodated in a single datagram. Hence we have created new requests which are compounded versions of selected requests. The advantages of these compounded requests and the subtle changes in semantics they introduce, are discussed below.

Lookup Path: Experiments show that the LOOKUP request is the most executed of all requests. The LOOKUP request implicitly shifts the burden of pathname traversal logic to the client and consequently generates a lot of extra traffic. While this approach is reasonable for normal clients, it is unacceptable for resource starved clients. For this purpose our implementation introduces a new request called LOOKUP-PATH. The pathname is included in the request and the file handle corresponding to the file is returned. Restrictions that are imposed include use of only absolute path names and conforming to the separator syntax of the server (obtained during a mount).

Lookup-Read and Lookup-Write: Lookup-Read combines the lookup-path request with the read request so that read operations typical of workload I can

be satisfied with one or two network transmissions. Lookup-write is also similar and the truncate-on-write which is used by normal write requests is also allowed.

Lookup-Readdir and the Readdir Cache: The LOOKUP-READDIR request is very similar to the lookup-read request but is used for reading directory entries. This is particularly efficient for small directories. The most frequent directory operation is the readdir operation. Since most filesystem semantics do not provide any guarantee about the order in which the different files occur within a single directory, applications that do readdir based operations typically read all the entries of the directory sequentially. In order to optimize for such workloads while satisfying the constraints imposed by resource starved clients, whenever an application reads a directory entry, we do a read-ahead using two or three READDIR requests and cache the two or three UDP datagram worth of directory entries (usually around 50 directory entries) in a time-to-live (TTL) based cache. Since a resource starved client typically supports only one or two processes, there is very little chance for heavy cache contention. Also the size needed is very small (around 25 KB). Further, since the cache is TTL based, no background work is needed for cache management. Thus all the constraints of resource starved clients are satisfied while providing good performance for subsequent readdir requests. A similar cache implementation for files will not be efficient since the nature of file workloads is fundamentally different from directory workloads.

Security: In this paper we ignore sophisticated security requirements. Such an assumption is not unreasonable since resource starved clients are not expected to handle sensitive data.

3.2 The Remote Procedure Calls

Based on the discussion above, the following RPC calls comprise the interface of the distributed file system. Details of the data structures are omitted. They are similar to what is used in NFS. It may noted that a client need not implement all the calls if space constraints are severe, as the compound calls *lookup_** subsume the functionalities of the simple calls. Implementing all the calls are however desirable. It is also to be noted that the *create* and *delete* calls apply to both files and directories (unlike NFS), with a parameter specifying which.

Mount, lookup, lookup_path, readdir, lookup_readdir, read, lookup_read, write, lookup_write, create, setattr, delete.

4 Implementation and Results

The above design was implemented under Linux with the server implemented as a user level process. For the client, part of the implementation was in the kernel (as a Linux kernel module) and part in user space. Essentially, the kernel module was used to intercept file system calls and to divert them to the user process which implemented the remote file access using a standard RPC library. Since most of the implementation was in user space, no Linux-specific kernel features

were used. The client code size is about 120 KB with 30KB comprising the kernel module and 90KB the user space code. The maximum variable memory usage per request is restricted to twice the maximum size of an UDP packet, that is, 16KB. So a total of 136KB on an Intel system is required to implement the client. Lean-dfs has been tested with file accesses of different sizes and using both reads and writes. The results show that the file system performs adequately under different loads. As the table below shows, performance is quite good. The tests were run with 1 Ghz CPU PCs as a client and a server, with Redhat Linux 8.0. Read performance is better or comparable to the performance of a local file system. Sequential writes performance is worse than the local case because there is no delayed write and so write-through is implemented. The "Updatedb" benchmark builds a database from a file. It is an I/O intensive benchmark with both reads and writes. Again because there is no advantage of caching, the performance of lean-dfs is quite good. Finally the program "fstree" copies an entire file system. Given that there is read and write involved with no advantage for caching, the results are consistent with the earlier results. The poor results for a large filesystem, though not important for the environments we envisage, could not be explained adequately. It is to be noted that while there is no caching at the client, caching with read-ahead and write-behind is implemented at the server side. So the advantage of server side caching is available to lean-dfs. This reinforces the case that absence of caching and asynchronous operations at the client side is not a big disadvantage for sequential accesses.

5 Conclusions and Future Work

This paper has described the design and implementation of a simple distributed file system, lean-dfs, for an environment where the clients are resource starved. A distributed file system is seen as an useful abstraction for implementing a wide variety of communication needs of such environments. The design has emphasised on low memory usage and the lack of adequate power at the clients. The implementation has demonstrated the feasibility of such a system meeting the goals. A number of extensions and future work can build on this prototype. *Surrogate Servers* : By using surrogate servers for a group of resource starved clients, a number of potential benefits could be realized. The clients can communicate with the surrogate server using the protocol explained above while the surrogate server can communicate with the remote server using a much more sophisticated protocol. *Heterogeneous Set of Clients* : We have assumed all clients to be resource starved. A more realistic environment would involve both resource starved and resource rich clients interoperating together. Further they could be spread over different locations involving different network conditions and security requirements.

References

1. Levy, E., Silberschatz, A.: Distributed File Systems: Concepts and Examples. Computing Surveys (December 1990)

Table 1. Experimental Results

Sequential Read with 1KB per Read		
Read Size	Local Filesystem (in ms)	Lean-dfs (in ms)
1 KB	0.194	6.211
512 KB	514.88	510.14
1 MB	1033.46	1018.53
Sequential Write with 1KB per write		
Write Size	Local Filesystem (in ms)	Lean-dfs (in ms)
1 KB	0.16	0.61
512 KB	69.48	651.12
1 MB	139.42	874.11
Updatedb Performance		
No. of Objects	Local Filesystem (in ms)	Lean-dfs (in ms)
100	5.0	11.0
2000	17.0	43.0
33000	39.0	94.0
Fstree Performance		
Filesystem Size	Local Filesystem (in ms)	Lean-dfs (in ms)
90 MB	19,323	36,847
200 MB	78,263	114,361
900 MB	100,595	1944,310

2. Braam, P.J., Nelson, P.A.: Removing Bottlenecks in a Distributed File System. Proceedings of the Linux Expo (1999)
3. Sandberg, R., et.al.: Design and Implemenation of the Sun Network File System. Proceedings of the USENIX Summer Conference (1985)
4. Pawloski, B., et.al.: NFS Version 3 Design and Implementation. Proceedings of the USENIX Summer Conference (1994)
5. Pawloski, B., et.al.: NFS Version 4 Protocol. Proceedings of SANE-2 (2000)
6. Satyanarayanan, M., et.al.: CODA: A Highly Available File System for a Distributed Workstation Environment. IEEE Transactions on Computers (April 1990)
7. Satyanarayanan, M., Kistler, P., et.al.: Experience with Disconnected Operations in a Mobile Computing Environment. Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing (1993)
8. Atkin, B., Birman, K.P.: MFS: An Adaptive Distributed File System for Mobile Hosts. Technical Report Cornell University www.cornell.edu/batkin/doc/mfs/ps (2003)
9. Muthitacharoen, A., Chen, B., Mazieries, D.: A Low-Bandwidth File System. Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (2001)
10. PalmOS: PalmOS (Cobalt) Documents on Files and High-Level Communications. www.palmos.com/dev/support/docs/protein_books/memory_databases_files.pdf www.palmos.com/dev/support/docs/protein_books/highlevel_comms.pdf (2004)
11. French, S., et.al.: Exploring Boundaries of CIFS. Proceedings of the CIFS Conference (2002)