Federated File Systems for Clusters with Remote Memory Communication

Suresh Gopalakrishnan, Ashok Arumugam and Liviu Iftode
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{gsuresh,ashoka,iftode}@cs.rutgers.edu

Abstract

We present the design, prototype implementation and initial evaluation of FedFS - a novel cluster file system architecture that provides a global file space by aggregating the local file systems of the cluster nodes into a loose federation. The federated file system (FedFS) is created ad-hoc for a distributed application that runs on the cluster, and its lifetime is limited by the lifetime of the distributed application. FedFS provides location-independent global file naming, load balancing, and file migration and replication. It relies on the local file systems to perform the file I/O operations.

The local file systems retain their autonomy, in the sense that their structure and content do not change to support the federated file system. Other applications may run on the local file systems without realizing that the same file system is part of one or multiple FedFS. If the distributed application permits, nodes can dynamically join or leave the federation anytime, with no modifications required to the local file system organization.

FedFS is implemented as an I/O library over VIA, which supports remote memory operations. The applicability and performance of the federated file system architecture is evaluated by building a distributed NFS file server.

1 Introduction

There has been an increasing demand for better performance and availability in storage systems. As the amount of available storage becomes larger, and the access pattern more dynamic and diverse, the maintenance properties of the storage system have also become as important as performance and availability. The problem is that storage systems which respond well to all these criteria are hard to build, and therefore the solutions are always trailing instead of anticipating the demand. One approach, that has been proven beneficial in the past, both in terms of flexibility and development time, is to leverage previous development effort, by using existing solutions as building blocks of new ones. So far, the distributed file system research has exploited this idea only in part [1, 2].

We propose a loose clustering of the local file systems of the cluster nodes as an ad-hoc global file space to be used by a distributed application. We call this distributed file system architecture, a **federated file system(FedFS)**. The simplest way to describe a federated file system is as a per-application global file naming facility that the application can use to access files in the cluster in a location-independent manner. By contrast, the NFS [3] solution of cross-mounting remote file systems into local file systems will allow applications to access files through a *location-transparent*, but not *location-independent* manner (files have their location implicitly embedded in the pathname through the mounting directory). Using the proposed federated architecture, a distributed NFS server, where files can be transparently migrated among cluster nodes can be built across a cluster.

In addition to global file naming, the description of a federated file system must include dynamic reconfiguration, dynamic load balancing through migration and recovery through replication. The proposed FedFS should provide all these features on top of *autonomous local file systems*. Autonomy means that local file systems are not changed in order to participate in a federation, and no federation specific metadata is stored in the local file systems. To achieve

this FedFS I/O operations translate into local file system operations and the global file space metadata becomes soft state that can be stored in volatile memory of the cluster nodes. As a result, a local file system can join or leave a federation anytime, without requiring any preparation, and without carrying out persistent global state operations. In this way, local file systems can simultaneously operate in a stand-alone fashion, and as part of one or more federations.

A federated file system is created ad-hoc, by each application, and its lifetime is limited to the lifetime of the distributed application. In fact, a federated file system is a convenience provided to a distributed application to access files of multiple local file systems across a cluster through a location-independent file naming. Interestingly, a location-independent global file naming enables FedFS to implement load balancing, migration and replication for increased availability and performance.

Figure 1 shows examples of federated file system configurations on a four node cluster. A1, A2 and A3 are three different applications running on the cluster. Application A2 is distributed across nodes 1 and 2 and uses FedFS1 to merge the local file systems of these nodes in a single global file space. Similarly, A3 is distributed across nodes 2, 3 and 4 and uses FedFS2. In this example, the local file system of node 2 is part of two federated file systems. A1 runs only on node 1 and uses the local file system directly.

There is a significant body of research related to distributed file systems [4, 5, 3, 6, 7, 8, 9, 10, 1, 2, 11, 12]. Some recent projects include [13], the emerging industry standard DAFS [14] and wide area systems like [15, 16, 17, 18].

In our project, we combine two technologies: the federated file system idea, and the remote memory communication support. Remote memory communication is the key ingredient of the recently proposed system area networks such as the Virtual Interface Architecture (VIA) [19] and the new I/O interconnects such as InfiniBand [20]. FedFS is implemented as an I/O library over VIA and exploits the non-intrusive remote memory operations. Preliminary performance of the federated file system architecture is evaluated on a distributed NFS built with the FedFS library.

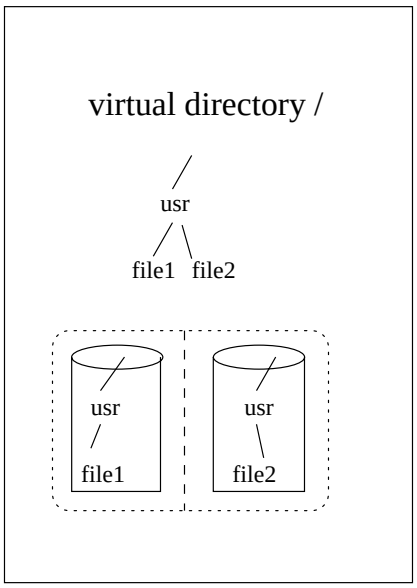

Figure 2: Virtual Directory /usr is formed by merging the /usr directories from the two nodes.(Section 2.1)

2 Federated File System Architecture

A federated file system is a distributed file system built on top of local file systems that retain their autonomy. Local file systems can simultaneously function, as stand-alone file systems or as part of FedFS. As in any federation, in FedFS, the file system functionality is split between the **federal layer (FL)** and the **local layer(LL)**. The LL is responsible for performing the file I/O operations on the local files as directed by the FL. Any local file system can be used as the local layer. The FL is responsible for global file naming and file lookup as well as supporting global operations such as load balancing, replication, coherence and migration.

2.1 Virtual Directories

FedFS aggregates the local file systems by merging the local directory tree into a single global file tree. A **virtual directory (VD)** in FedFS represents the union of all the local directories from the participating nodes with the same pathname. For instance, if a directory /usr exists in each local file system, the virtual directory /usr of the resulting FedFS will contain the union of all the /usr directories. Figure 2 shows a virtual directory created by merging the two local file systems.

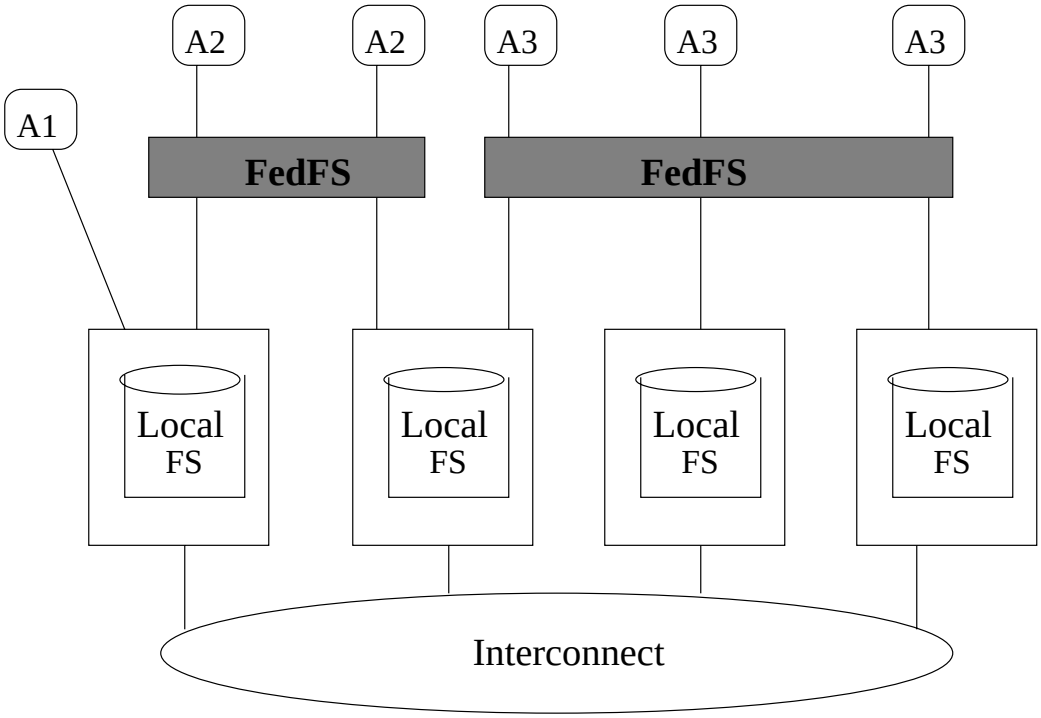

Figure 1: FedFS configuration example (Section 1)

One prominent advantage of this aggregation strategy is location-independent naming. Because the virtual directory is the union of the local directories with the same pathname, the pathname of a file indicates the virtual directory but does not provide any information about where it is located. Therefore, in FedFS, files can naturally migrate from one node (local directory) to another without changing their pathname (virtual directory).

The content of a virtual directory is discovered on demand (whenever it is necessary to solve a lookup) by performing a directory merging or **dirmerge** operation. To amortize the dirmerge operation over multiple directory accesses, the virtual directory content calculated by a dirmerge is cached in volatile memory of the manager. The manager may however discard a VD if it runs low in memory in which case its content will be regenerated by another dirmerge when the next access occurs.

To allow file migration without requiring virtual directory modification, we associate with each pathname (virtual directory or file), a **manager**. The nodes where the corresponding pathnames are present in the local file system are called **homes**. The manager will be determined by applying a consistent hash function to the pathname. For a file, the manager is responsible for keeping the home information. For a virtual directory, the manager is responsible for the VD content which can be either cached or calculated using the dirmerge operation.

2.2 Dirmerge and Directory Tree Summaries

The dirmerge operation is initiated by the manager to calculate the content of a virtual directory. To perform a dirmerge, the manager has to send a `readdir` request to all the nodes of the cluster that may have that directory in their local file systems. Obviously, this is not a scalable solution, although we expect to perform it quite rarely.

To make the dirmerge operation scalable, we propose each node to generate a *summary of the directory tree* and pass it to every other node when the cluster is first established or when the node joins the cluster. The directory tree summary will be calculated using the Bloom filters [21] and will include only the directory tree without the files.

If a dirmerge is necessary, the manager node will use the summaries to determine which nodes may have that directory in their local file systems and direct the readdir request only to those nodes. Since Bloom filters generate only false positives, dirmerge is guar-

anteed not to miss any node which has the directory.

Updating the directory tree summary is an expensive operation but it is possible to perform this operation infrequently (for instance, only when a number of changes to the local directory tree have been performed). Whenever a new directory is created, only the summary of the manager of the corresponding virtual directory must be updated. Therefore, instead of recalculating the summary and sending it to every other node, a simple update to the manager of the newly created directory suffices. Directory deletions will only create additional false positives.

We did some preliminary analysis of bloom filters to confirm the feasibility of using them for summaries. The tests yielded extremely good results. The performance of bloom filters depend on the number of bits used for the summary as well as the number of hash functions used.

We did experiments with a summary size of 2^{20} bits (128KB) and with 185060 valid keys and 109526 invalid keys (keys representing unique pathnames). With 3 hash functions, there were 208/185060 collisions and 477/109526 false positives. With 5 hash functions, we got almost zero collision (1/185060) and very low false positives (8/109526).

The cost of each hash function is approximately 6xlen+35 instructions, where len is the number of bits in the key. Thus, with average pathname length of 40 characters (320bits), each hash takes approximately 2000 instructions, 5 hash functions will take 10000 instructions. This corresponds to an actual time of around 10 microsecs on a 1 GHz processor and 1 instr/cycle, which is at least 50% faster than the fastest round-trip time over the current VIA implementation to check for the directory on the remote node.

2.3 Directory Table

Under the FedFS architecture (see figure 3), file lookup always requires an extra access to the file manager to determine the home of the file. To eliminate this extra step in the common case, we propose to add a *directory table*(DT) to each node which will act as a cache of virtual directory entries for the most recent file lookup accesses.

In the DT, an entry must contain the full pathname of the file and not just the local name as it is stored in the virtual directories. (This is analogous to the TLB, which caches entries for multiple page tables

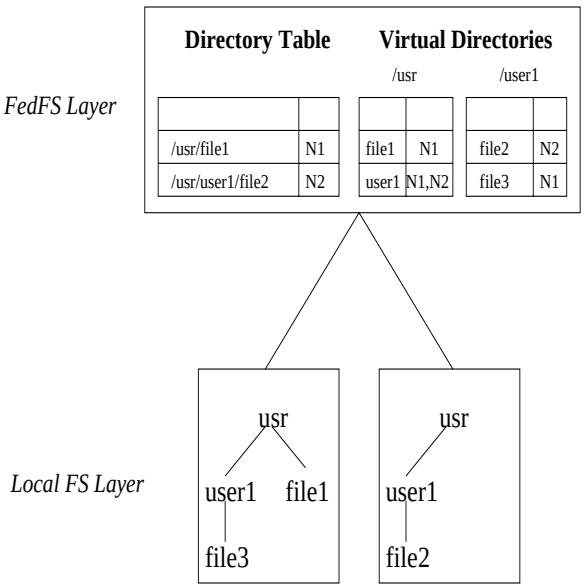

Figure 3: FedFS Architecture (Section 2.3)

and hence virtual page numbers must be accompanied by the process id.) The access to the directory table will be performed using a hash on the full pathname. However, the open file table may contain an index in the directory table of the local node or directly of the home node of the open file to avoid hash function calculation on each file access.

3 Federal Layer Operations

In this section, we describe the operations performed by the federal layer namely file lookups, file migration and replication and dynamic reconfiguration.

3.1 File Lookup Operation

The lookup operation is performed to locate a file i.e. determine the home for the file from its pathname. Figure 4 illustrates the four possible paths a lookup operation can take.

1. *Any node* performing a lookup will first search its local directory table for a previously cached entry. If there is a hit in the DT (likely if file accesses exhibit good temporal locality), the lookup completes at the local node.

2. If there is a miss in the local DT, the lookup operation will contact the manager of the file.

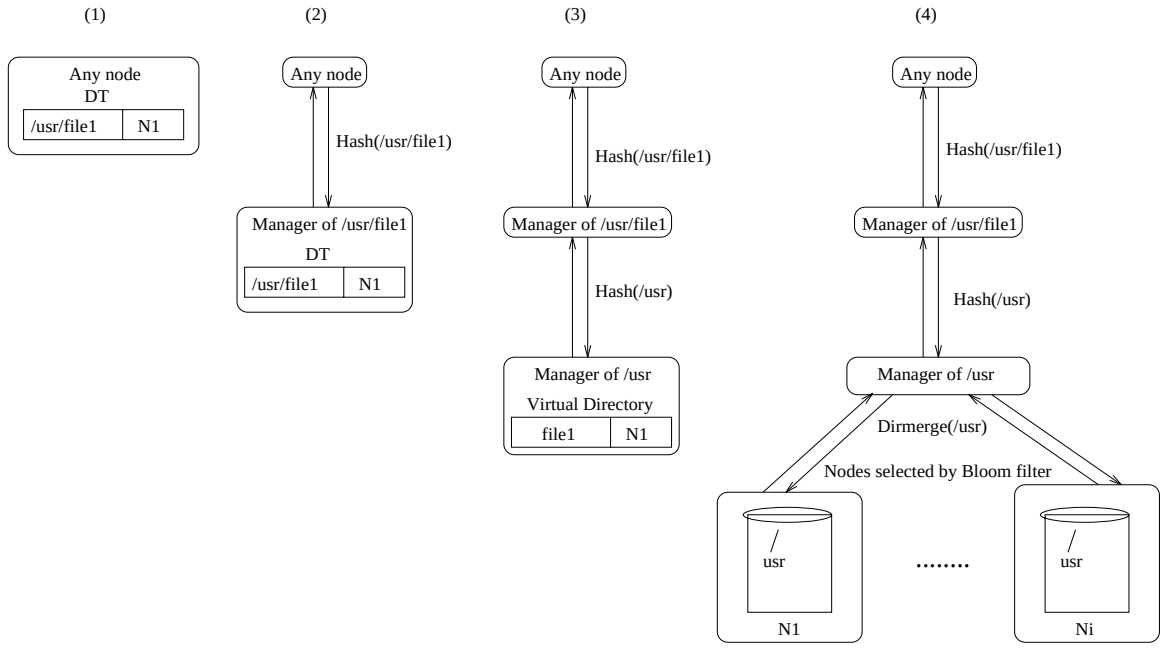

Figure 4: Various Lookup cases (Section 3.1)

The manager is determined by a hash on the pathname. The manager refers to its DT to find the home of the file and if found, the lookup terminates.

3. If there is a miss in the DT at the file manager, the lookup operation contacts the manager of the file's parent directory. The parent directory is easily obtained from the pathname and the parent's manager is located by using the hash function. If the manager of the parent has the virtual directory cached, the lookup completes and the home of the file is returned.

4. Finally, if the virtual directory is not cached, the parent directory calls for a dirmerge operation to construct the virtual directory. As explained in the previous section, we use bloom filters to contact only the subset of the nodes that are likely to have that directory in the local file system.

Lookup operations are expected to be fast in the common case. The cost of querying the file's manager, querying the parent's manager and doing a dirmerge at the parent's manager are one time costs, easily amortized over multiple lookups.

3.2 File Migration and Replication

File migration and replication are enabled by the location independent file naming using the virtual directory architecture and additional level of indirection involving managers in the lookup path.

Whenever the migration policy decides to move a file, the file is added to a list at the target node. For each file in the list, the file is transferred and the file's manager and parent's manager are updated with the information about the new location. This mechanism ensures that migrating a file does not disrupt service of that file. When the home of a file changes due to migration, some of the cached DT entries in other nodes become stale. They are not invalidated these eagerly, rather the nodes use the stale information and discover that the file is no longer present. Then, the manager is queried again to find the new home. This lazy mechanism presents additional overhead if a lookup happens on a file that was deleted, but this is not a common case.

Under our replication policy, for each file, two coherent replicas (primary and secondary) are maintained. On a lookup, the manager returns the primary node as the home of the file. If the primary becomes unavailable, the manager can redirect subsequent lookups to the secondary. When one of the copies become unavailable (node leaves or crashes), the manager create another copy.

3.3 Dynamic Reconfiguration

In FedFS, nodes can join the federation to increase the file set and storage, as well as leave the federation. When a node joins FedFS, manager responsibilities for some files and directories are transferred to it. The hashing mechanism to locate managers is able to accommodate this change, because we incorporate the number of nodes in the hash function. The new node will also send its summary information to all the nodes. When a file lookup occurs at the new node, the query will reach the manager of the parent. If a new node summary arrives after the last dirmerge, the parent will perform incremental dirmerge involving only the new node and the file becomes visible as part of the global space.

When a node leaves, the files and directories for which this node was the manager, are handed off to other nodes. Files for which the leaving node was one of the consistent replica locations now have to be replicated on another node.

4 Prototype implementation

A prototype implementation of the FedFS has been built as a user level library in Linux that exports the standard file system interface. The FedFS communication library is built using VIA. The Bloom summary (4Kb per node) is generated using 4 hash functions.

The first application used to test FedFS with is a user level NFS server (NFSv2) on Linux. An NFS server can serve only local files, below the exported mount point. An NFS server linked with FedFS, called DNFS (Distributed NFS), can distribute its files on all the nodes in the cluster, and serve them out of any of the nodes. All the results presented are based on this DNFS server. The file placement policy used in the implementation is to collocate a file or directory with its manager. File migration mechanism has been implemented, but a policy is not in place yet.

All experiments were performed in a cluster of 8 Pentium II 300MHz dual-processors, each with 512KB cache, 512MB memory and 2 SCSI disks each (one 3.6GB IBM disk and one 8.3GB Quantum disk) and running Linux 2.2.14 kernel (smp version). Each node has a SMC Epic100 Ethernet card and a Giganet VIA card used only for intra-cluster communication. Client-server communication uses the Ethernet, the server-server communication uses the Giganet. The cache maintained at each server is 128MB.

Operation	Percentage (%)
Lookup	46
Read	34
Write	17
Create	2
Remove	1

Table 1: SPEC97 operation mix used in the experiments (Section 4)

Except for the micro benchmarks, the SPEC97 SFS or LADDIS NFS benchmark [22] was used as the driving client to test the performance of DNFS. The mix of file operations used in our experiments is shown in Table 1.

4.1 Micro-benchmarks

Table 2 lists some micro benchmarks that show the cost of FedFS operations. The first column is the latency of calls made to local file system (ext2). FedFS "0 hop" column shows the latency of calls made through FedFS, where all operations are local. The other columns show the latency of calls when the files are created on a remote node or when the managers are located remotely and messages need to be exchanged. The worst case is when up to 6 messages are exchanged for an operation (file is created on a remote node, its manager is on another node and the parent directory's manager is on a different node). The blank entries are situations that did not happen in the test runs.

The cost for remote operations has three components. First is the latency due to network communication. On a Gigabit network, this latency is of the order of tens of microseconds. Second is the queuing delay in the remote node. To avoid serializing parallel requests, request messages are first queued by a network polling thread and then picked up by the protocol thread. The network and queuing delay for a message exchange is roughly 200 μs. The final component is the operation latency at the remote node.

For creat, first a lookup is done, and the operation is performed only if the file is not found. After creating, the parent directory has to be updated and this might result in a message if the parent's manager is on a remote node. For mkdir, an additional message may be required to update the summary if the manager

Call	Local (μs)	FedFS (μs)					
		0 hop	1 hop	2 hops	3 hops	5 hops	6 hops
Creat	89	140	-	522	-	1179	-
Mkdir	164	175	-	-	820	-	1420
Read	105 (14)	105 (11)	452	-	-	-	-
Write	19	75(16)	305	-	-	-	-
Unlink	25	90 (62)	- (339)	570	792 (664)	-	-

Table 2: Latency of operations from micro-benchmarks. Local is the local ext2 file system. A FedFS operation may involve multiple messages depending on the location of the managers of the file/dir or it's parent. (Section 4.1)

Call	NFS (ms)	DNFS (ms)
Lookup	0.50	0.52
Read	1.62	1.62
Write	1.28	1.51
Creat	1.16	1.33
Remove	0.65	0.72
Average	1.03	1.08

Table 3: Latency of some operations from macro-benchmark (SPEC97). NFS is a regular user level NFS server, DNFS (Distributed NFS) is the NFS server linked with FedFS.(Section 4.1)

of the new directory is on a remote node. For reads and writes, the cases of cache miss and hit are shown. For unlink, the numbers in parentheses show the case where lookup had a hit in the local Directory Table.

Table 3 shows the overhead of FedFS running macro-benchmarks with DNFS server and SPEC97 client. In this test configuration, there is only one server and one client. The client requested load is 50ops/sec, for a duration of 300 secs, resulting in 15000 operations. Around 19500 files are created in 650 directories, and the total data set size is 495MB. After discounting the client/server communication latency (which uses UDP over the Ethernet), the latency is only marginally higher on DNFS compared to NFS.

4.2 DNFS Performance

The next experiment compares the performance of DNFS against NFS. First, the SPEC97 benchmark is set up with a single node running the regular NFS server and four clients mounting a single volume from the server. Next, with DNFS, the clients mounted the same volume from four different servers, while accessing the same file set. The file set is now distributed across the nodes, but they are all accessible

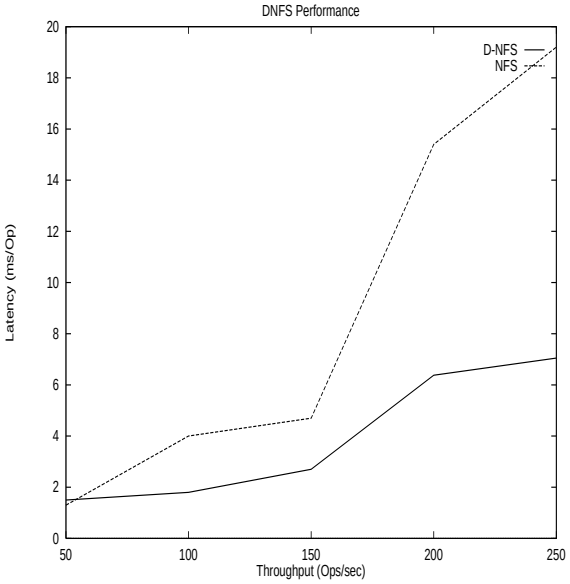

Figure 5: DNFS Performance: The graph shows average operation latency against load offered by the clients. NFS is the regular NFS server, DNFS is NFS linked with FedFS, running on four nodes.(Section 4.2)

from a single mount point due to FedFS. The file set and data set size created and accessed are similar to the previous experiment, but proportional to the requested load. Figure 5 shows that, as expected, DNFS scales better than regular NFS, since the same load is now spread across multiple servers, while serving the same file set.

The above shows the scaling with respect to clients and client loads. FedFS scales with respect to server configurations also. Adding more nodes to FedFS only increases the aggregate storage and bandwidth it can deliver, without additional communication costs. This is because almost all FedFS operations involve communication between two nodes - a requesting node and the home or the manager. The only operation that involves more than three nodes is the dirmerge operation, which is performed only once per directory in the entire FedFS run time.

5 Current issues

The initial proof-of-concept prototype has a few drawbacks that we are currenty addressing. The messaging cost of 200 μs is quite large compared to what is achievable using Giganet VIA. We have since then redesigned the communication layer and significantly reduced this cost. Another factor affecting the server performance is the "double buffering" phenomenon. File data is cached in the FedFS layer as well as in the kernel buffer cache. This causes contention for memory and to avoid swapping, the cache size has to be quite small. We are currently making changes to use the direct I/O feature available in the latest kernels to avoid the double buffering.

References

[1] David Mazihres. Self-certifying file system. PhD thesis, MIT (May 2000).

[2] M. Ji, E. Felten, R. Wang and J. P. Singh. Archipelago: An Island-Based File System For Highly Available And Scalable Internet Services. Proc. of 4th USENIX Windows Systems Symposium (2000).

[3] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. RFC 1094 (1989).

[4] M. Satyanarayanan. A Survey of Distributed File Systems. Research Report CMU-CS-89-116, CS Department, Carnegie-Mellon University (1989).

[5] Peter Honeyman. Distributed File Systems. Distributed Computing: Implementation and Management Strategies, ed. Rhaman Kanna, pp. 27-44, Prentice-Hall (1994).

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli and R. Y. Wang. Serverless Network File Systems. Proc. of 15th SOSP (1995).

[7] Chandramohan A. Thekkath, Timothy Mann and Edward K. Lee. Frangipani: A Scalable Distributed File System. Proc. of 16th SOSP, (1997).

[8] Bjorn Gronvall, Assar Westerlund and Stephen Pink. The Design of a Multicast-based Distributed File System. Proc. of OSDI (1999).

[9] Lily B. Mummert, Maria R. Ebling and M. Satyanarayana. Exploiting Weak Connectivity for Mobile File Access. Proc. of SOSP (1995).

[10] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton and Jacob Ofir. Deciding when to forget in the Elephant file system. Proc. of SOSP (1999).

[11] M. Rosenblum and J. K. Ousterhout. The design and implementation of a Log Structured Filesystem. Proc. of 13th SOSP (1992)

[12] J. H. Howard. An overview of the Andrew File System. Proceedings of the USENIX Winter Technical Conference (1988).

[13] Darrell C. Anderson, Jeffrey S. Chase, Amin M. Vahdat. Interposed Request Routing for Scalable Network Storage . Proc. of 4th OSDI (2000).

[14] Direct Access File System. http://www.dafscollaborative.org

[15] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. Proceedings of the 9th ASPLOS (2000).

[16] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of the 18th SOSP (2001).

[17] Frank Dabek, M. Frans Kaashoek, Robert Morris (MIT) and Ion Stoica. Wide-Area Cooperative Storage with CFS. Proceedings of the 18th SOSP (2001).

[18] Athicha Muthitacharoen, Benjie Chen and David Mazieres. A Low-Bandwidth Network File System. Proceedings of the 18th SOSP (2001).

[19] Compaq Corporation, Intel Corporation and Microsoft Corporation. Virtual Interface Architecture Specification 1.0. http://www.viarch.org

[20] The Infiniband Architecture. http://www.infinibandta.org

[21] B. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM (July 1970).

[22] B. E. Keith and M. Wittle. LADDIS: the Next Generation in NFS File Server Benchmarking. Proc. of USENIX Summer Technical Conference (1993).