H-0141 October 10, 2002
Computer Science

IBM Research Report

DSF - Data Sharing Facility

Zvi Dubitzky, Israel Gold, Ealan Henis, Julian Satran, Dafna Sheinwald
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

DSF - DATA SHARING FACILITY

Zvi Dubitzky

Israel Gold*

Ealan Henis

Julian Satran

Dafna Sheinwald

IBM Research Laboratory in Haifa

(* now with SANGate Israel)

Abstract

This paper presents DSF - a new serverless distributed file system, aimed to improve scalability. Scalability is obtained by moving traditional file system functionality to lower (disk) levels and by using a dynamic file management assignment policy to improve load balancing.

1. Introduction

Disks are slower than processors, and modern networks performance is improving at a higher rate than that of disks. Moreover, the fast processors and networks impose an increasing average and peak demands on storage and file systems.

A typical centralized network file system is based on a dedicated file server that satisfies storage access requests, manages file system metadata and maintains a cache. These tasks make the file server a performance and reliability bottleneck, and the centralized solution does not scale well.

The Data Sharing Facility (DSF) presented here is a scaleable non-centralized ("serverless") distributed storage access system, where storage, cache and control is distributed over cooperating workstations.

Existing systems provide limited answers to the growing storage access demands. NFS [1,2] is a remote file access protocol that provides a weak notion of cache consistency. Its stateless design requires clients to access servers frequently to maintain consistency. NFS4 [13] introduced client caching and state-based protocol. AFS [3] provides local disk caching and consistency guarantees, but it does not implement a native file system. It has a global namespace, but a single centralized server manages each mountable volume. The VMS Cluster file system [4,5] offloads file system processing to a group of individual machines that are members of a cluster. Every cluster member runs its own instance of the file system code on top of a shared physical disk, with synchronization provided by a distributed lock service. The shared physical disk is accessed either through a special purpose cluster interconnect to which a disk controller can be directly connected, or through an ordinary local area network such as Ethernet and a machine acting as a disk server. The Frangipani clustered file system [6] improves upon this design by replacing the shared physical disk with a shared scaleable virtual disk provided by Petal [7]. Petal consists of a collection of network-connected servers that cooperatively manages a pool of physical disks. To a Frangipani cluster member, this collection appears as a highly available block level storage system that provides large abstract containers, which are globally accessible by all Frangipani cluster members. IBM's GPFS has similarities with Frangipani in its log-based recovery. However, GPFS does not scale well due to the use of a centralized lock server. xFS [8] attempts to distribute all aspects of

file service over multiple machines across the network to provide high availability, performance and scalability. However, xFS management distribution policy is static, and its recovery mechanism is complicated (log based).

The DSF presented here is closest to xFS among existing systems. Functionally DSF differs from xFS by providing dynamic distribution of file management.

It has also different functionality in the components. Through this new functionality it achieves a simplified structure, better scaling and simplified metadata recovery, all of which potentially improves performance and reliability.

2. DSF design principles

The DSF design is depicted in Figure 1.

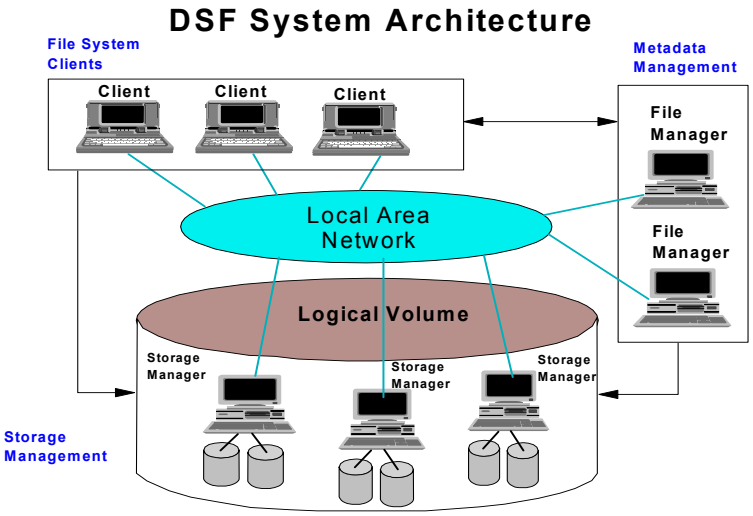

Figure 1 DSF - System Architecture

The System component and functions are:

- DSF Client - runs on some workstation and provides access to DSF files and directories. The DSF Client maintains a memory cache of data blocks accessed by applications on the Client workstation. The Client accepts file system requests from user programs, sends data to Storage Managers on writes, forwards reads to File Managers on cache misses, and receives replies from Storage Managers or other Clients. It also answers forwarding requests from File Managers by sending data to other Clients.

- DSF File Manager - manages the metadata and cache consistency for a subset of DSF files. To provide scalable service DSF splits the management of its files among several File Managers. The Manager of a file controls two sets of information about it, cache consistency state and file structure metadata blocks. Together these structures allow the File Manager to track all copies of the file's data blocks. The File Manager can thus forward Client read requests to other Clients thereby implementing cooperative client caching.

- DSF Storage Manager - stores and maintains data and metadata blocks on its local disks. The Storage Manager reacts to requests from File Managers by supplying data to Clients which have initiated I/O operations. DSF Storage Managers contain the intelligence to support DSF Logical Volumes.

- DSF ***Logical Volume*** - consists of ***a collection of logical disks*** that span multiple Storage Manager machines and provides abstract interface to disk storage with a rich set of recovery properties. The logical volume hides the physical distribution of its logical disks so that new disks and Storage Managers can be incorporated into the system dynamically and without interrupting system operation, thereby increasing storage capacity and throughput. The storage system may be further reconfigured by moving disks between Storage Managers, to match different working environments and workloads. Expanding the storage space (volume scalability) can also be done without interrupting system operation

DSF performance and scalability is achieved by the following design elements:

- Separation of storage from file management.
- Distribution of storage management over multiple machines

- Dynamic distribution of file and metadata management across multiple machines

  Caching and metadata management can be done on a machine that is different from the one storing the data.

- Cooperative caching

  Client machine memories are treated as one global cooperative cache. Clients are able to access blocks cached by other clients, thereby reducing Storage Managers load and reducing the cost of local cache miss.

- Lack of dedicated machines

  This eliminates source bottlenecks. Any machine in the system, including one that runs user applications, can be made responsible for storing, caching and managing any piece of data or metadata. Furthermore, any machine in the system can assume the responsibilities of a failed component.

- Extensibility - machines can be added to the system

- Freedom to configure the file system.

  DSF can be configured to match different system environment depending on machine memories and CPU speeds. DSF can have multiple configurations ranging from a "small office system", where the file system is shared between two machines and only one is responsible for storing the data, to a large "clustered system" of hundreds machines, where each machine is made responsible for storing, caching and managing parts of the file system.

- Use of logical volumes

  DSF logical volume can be used to dynamically reconfigure the storage subsystem without interrupting file system operation. The support for transactional operations over multiple disks improves the performance of file operations that require atomic multi-block writes, like file sync and directory

operations. The new allocateAndWrite technique removes the need to allocate a new block prior to its writing to disk

3. DSF mechanisms

3.1. DSF Logical Disk - Introduction

The Storage Manager maintains each of the plurality of disks physically attached to it as a *logical disk*.

The idea of a *Logical Disk* that binds logical block addresses to physical block address via a *translation table* is not new. At MIT **[13],** HP **[14, 15],** DEC **[6,7],** Princeton U, and more, prototypes of the logical disk architecture are built since the early 1990's.

With the logical disk approach, the disk storage is partitioned into fixed size block *spaces*, each made of several consecutive sectors. The user application (file system, or Data Base, etc.) refers to its data as partitioned to (logical) blocks, each is worth of the size of slightly (some 30 bytes) smaller than a block space, and it associates *logical addresses* to its blocks. The Storage Manager maintains a *translation table* that converts each logical block address to physical disk address, which is the sector address of the first sector in the block space that accommodates the most recently stored contents of the logical block. The Storage Manager also maintains an *allocation bitmap* where it records the availability of the block spaces on the disk.

Dynamic change of these tables provides for the ***stable storage*** feature: only after an available block-space is allocated and the new contents of a logical block are safely stored into it, is the block-space that accommodated the old contents released for use by further block stores, and the *translation table* is updated.
With such a scheme, a write that fails due to a faulty disk-sector is retried at another block space (transparently to the caller of the write operation, which only refers to blocks by their logical addresses), and if a failure occurs in the midst of block writing, the old contents of the block can be fully recovered. Thus, successful write of a block ends with the logical address of that block bound to the fresh contents of the block, stored sound on the disk, and a non-successful write -- to the old contents. A write operation never ends with indefinite contents of a block - therefore the attribute *stable*.

Without the *translation table*, no block can be found on the disk, and without the *allocation bitmap* available block-spaces are hard to find. To withstand a power failure, it does not suffice to keep these data structures in volatile memory; periodically, these data structures must be flushed to disk. Furthermore, every block write must include sufficient information that can be used, during recovery, to redo the updates made to the *translation table* and the *allocation bitmap*, which we show in the sequel.
The logical disk scheme we used for DSF simplifies the data structures manipulation, improves performance of ordinary and recovery operations, and further extends the services provided by the logical disk, to include operations traditionally done by the file systems. It also allows more than a single logical disk user to use it without having to coordinate operations. *DSF logical disk* provides also transactional store of multiple blocks, over multiple disks.

Our mechanisms provide the following benefits, most of which we have not seen in none of the existing implementations of logical disks:

1. Allow I/O in blocks made from several consecutive sectors (thereby allowing atomic multi-sector write). That is, the disk can be managed in blocks, larger than its sectors, whose size is determined when the disk is formatted as DSF logical disk.
2. The physical disk, on which the DSF logical disk is implemented, contains all the information needed for its management, and thus the disk is **easily movable from one host** to another, without calling for a total re-configuration.
3. On each block write, **at no extra I/O cost**, the update of the *translation table* is stored to disk as well.
4. On each block write, **at no extra I/O cost**, the update of the *allocation bitmap*, which records the availability of block spaces, is stored to disk as well. Preallocation of several blocks, and the storage of the updated *allocation bitmap*, is also not needed, as opposed to the usual practice, and hence no leak of space occurs when the storage server fails and the preallocated blocks are lost.
5. It **provides *allocation* and *deletion* of blocks**. This allows multiple users of our DSF logical disk to allocate blocks without the need to synchronize their requests and protect against collisions. Moreover, our allocation and deletion schemes withstand cache failures.
6. It provides **soft-write, commit,** and **abort** operations which enable the two-phase commit needed for atomic multi-block stores (on single or multiple disks).
7. Consecutive stores of blocks (not necessarily in one chunk) make the disk arm move mostly forward; once in a while, the arm is reset all the way

backward, and then again it moves forward for many stores. Although the stores are not necessarily adjacent on disk, the one directional, rather than random, move of the arm gives better performance (as in log structured file systems).

8. Checkpoint of our scheme's data structures (store to disk of the *translation table* and *allocation bitmap* and a few integer variables) takes place when the arm moves backward, or earlier, at the DSF logical disk's convenience; i.e., **timing for checkpoint is rather flexible**. Checkpoints can be done **succinctly** by identifying the components that changed since the last checkpoint, and can even be made **piecemeal**, in small parts, one at a time.
9. **Recovering** from cache (power) failure, *DSF logical disk* reconstructs its in-memory data structures, **bringing them to the very same state they were in immediately prior to its failure**, faster than any previous work, known to us, in this area. The time consumed is **linear** in the number of write operations that took place since the last checkpoint. Besides avoiding scanning the whole disk, the read operations needed for the recovery are ordered such that the arm only moves forward.
10. The stable store mechanisms **can co-exist with the conventional store in place** mechanism on one disk: part of the disk is managed through translation table and allocation bitmap, and the other part is managed as a simple disk.
11. The implementation of our scheme is rather **simple and suits modern disk controllers**.

3.2. Formatting the disk

Some of the space of the physical disk is reserved for describing general physical and logical parameters, like size of disk sector, number of sectors, number and size of block spaces, range of logical addresses supported by the disk, etc.. Space is also reserved for checkpoints of the data structures used to manage the *DSF logical disk*. The rest of the space is partitioned to *block-spaces*. An *allocation bitmap* is constructed for the disk, associating one bit with each block-space thus defined. Initially, all the block-spaces are free and accordingly all the bits of the *allocation bitmap* are set to 0.

3.3. Allocation of Block-spaces

When a block-space is needed for storage, one is allocated from the free block-spaces on the disk. Our scheme records the physical sector address of the last block-space allocated, and looks for a free block-space from that address forward. As explained in the sequel, our scheme employs block chaining: the blocks are stored with a forward pointer, yielding a **forward linked list** made from all the blocks stored. This allocation, store and link forward process

continues until no free block-space is found whose address is higher than the last block-space allocated. When this happens, a *checkpoint* is called, whereby the *DSF logical disk* stores its data structures to disk, and the allocation process resumes from the free block-space of lowest address, creating a new forward linked list of stored blocks. All along the process of store and link forward, between successive block stores, the disk arm moves in one direction: forward. The one directional move of the arm gives good performance, as in log-structured filesystems.

3.4. Storage Management Data Structures

In addition to the *Translation Table* and the *Allocation Bitmap,* our *DSF logical disk* also maintains the *Pass Number*: a counter of the number of times that the disk arm completed move-forward-and-store passes. This equals the number of checkpoints done thus far; the *First Available Block-space*: a pointer to the first, in address order, available block-space when a checkpoint takes place; and the *Next Available Block-space*: a pointer to the available block-space which will be stored to the next storage operation.

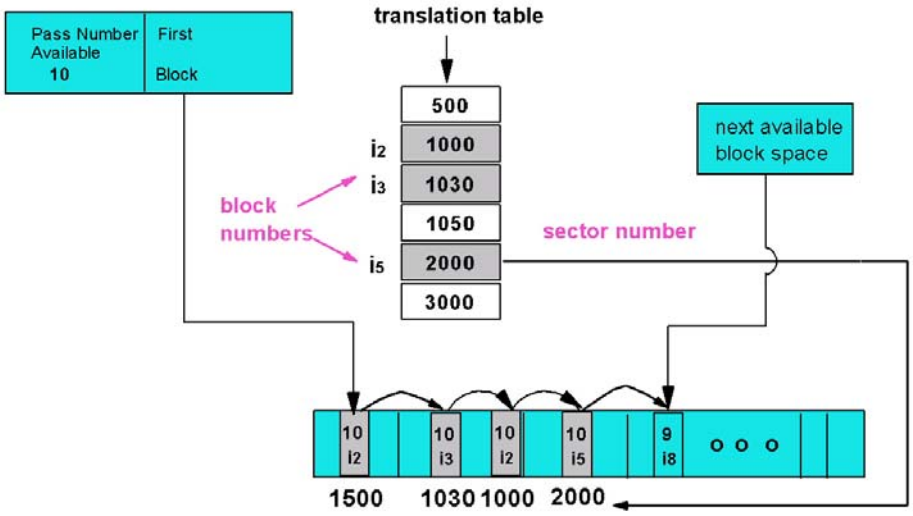

Figure 2 Translation Table and block chain operations

3.5. DSF Logical Disk Operations

The **Read (*address*)** operation is straightforward. If the address is in the range of the metadata addresses, then *address* is logical, and the *translation table* is consulted, and the contents of the block whose logical address is *address* is returned. If *address* is in the range of the conventional store, then the contents of data block whose physical address is *address* is returned.
The **Write (*address, contents*)** operation is also straightforward. On the conventional part of the disk (where a write overrides old data) this is the ordinary store of *contents* into the space whose physical address is *address*. In the meta-data part, this is a stable store, as described above: first a new block-space is allocated, into which *contents* are stored, and associated with logical address *address*. Then, the translation table is updated, and finally, the block-space that used to hold the previous contents of the block whose logical address is *address,* is released.

The non-conventional disk operations are:

Allocate and Write (*contents*): For regular blocks it means getting a free block, allocating it and returning the address to the caller. For stable-storage (metadata) blocks it means also finding a logical address, by looking in the *translation table* for an entry that is mapped to NULL, and a free block-space, and then continue as with Write, and return also the logical address allocated.

Write (logical address, contents) For stable-storage blocks it means getting the next free block in the chain, writing the content and updating the in-memory translation table.

Delete Blocks (*i1,i2,...*): For stable-storage blocks it involves deleting the binding of blocks of logical address *i1,i2,...* with any stored contents, and for all making the spaces used to hold their contents available for further block stores. For stable store blocks, the *DSF logical disk* stores a special block with deletion information. This block occupies a block-space only until the next checkpoint operation, at which time the deletion information is stored to disk in the form of the stored updated tables, and the block-space that accommodates the deletion information becomes available. For regular blocks it involves only making the blocks available for reallocation.

Softwrite(Transaction-id, logical-address, contents): Allocate a block-space into which contents of logical block, associated with logical address *l,* are stored, but the old contents still remain on the disk. An extension of the translation table makes a note of this ambiguity. Once **Abort (Tid)** is issued, the new contents that pertain to transaction Tid are removed (i.e.,

the block-space that accommodate them becomes available), along with the removal of the ambiguity notification. Once **Commit(Tid)** is issued, the analogous removal of the old contents takes place.

3.6. CheckPoint

CheckPoint can take place at any time. It is mandatory, though, when no available block-space is found beyond *Next Available Block-space*. In *CheckPoint*, the following items are flushed from volatile memory to a preallocated space on disk that is dedicated for the checkpoint:

1. *Translation Table* and *Allocation Bitmap*.
2. *First Available Block-space*, which is the first (associated with lowest sector address) block-space marked free by the *Allocation Bitmap* at the time of *CheckPoint*.
3. *Pass Number* after increment.

The store of *Translation Table* and *Allocation Bitmap* dominates the amount of time consumed by CheckPoint. This store operation can be done efficiently by partitioning the data structures to segments of sector size, and each time one of these data structures is updated, the relevant segment is marked. Then, on CheckPoint, only the updated segments are stored to disk - each to one disk sector. This way, if checkpoints are frequent, due to very small number of free block-spaces, the updates between successive checkpoints are very few, and the checkpoint process is very short. When checkpoints occur infrequent (abundance of free space) the overhead is negligible compared to the ordinary activity.

On CheckPoint, the value of *Next Available Block-space* is set to be *First Available Block-space*; thereby a new pass of move-forward-and-store takes off. Immediately following disk formatting, a first CheckPoint, of all the initial values of the data structures, takes place. (This generalizes the recovery process).
If no free block-space is found when CheckPoint takes place, an error message is issued.

CheckPoint in time-bounded segments. When CheckPoint takes place, *DSF logical disk* ceases to provide service, because all its data structures are locked until the store to disk is complete. As this may have a negative effect on response time we suggest here a simple scheme for making CheckPoint in small, time-bounded segments. Once the store-and-link-forward reaches a point when CheckPoint should start, copies of *the Translation Table*, *Allocation Bitmap*, and *Next Available Block-space* are made in main memory, *Pass Number* is incremented, and then the ordinary operation of *DSF logical disk* continues. Then, in between operations, when it is not busy, "on its leisure" (in other

words, by a thread with a low priority, for example) the *DSF logical disk* stores, segment by segment of the copies, to special dedicated place on the disk, (that it always alternates between 2 checkpoints). Because the store is from the copies, it does not block the ordinary work with the original data structures. When the copies of the tables are all stored to disk, the *DSF logical disk* also stores *Pass Number* and the kept value of *Next Available Block-space*, as the checkpointed value *First Available Block-space*. From that moment on, the newly stored data structures, plus all the block-spaces stored to since that CheckPoint started (from the block-space pointed at by the stored value of *First Available Block-space*), suffice to recover all data structures, in case they are lost on a power-failure. The scheme of store-and-link-forward may continue, and it even may wrap around the lowest disk addresses and then continue forward, but it should not pass over the block-space pointed at by the stored value of *First Available Block-space*. If a failure occurs while this incremental CheckPoint takes place, the information stored to disk on last CheckPoint, plus all the block-spaces stored to since that CheckPoint (which are uniquely identified by their being linked forward, starting with the block-space pointed at by *First Available Block-space* stored at last CheckPoint, and by their *Pass_Number* field containing the *Pass Number* stored in last CheckPoint, or a value greater than it by 1) suffice for a full recovery of *DSF logical disk*'s in memory data structures.

3.7. Migration of Disks:

On a storage system where each disk is only attached to a single host, failure of that host makes the disks attached to it inaccessible. When host failures last too long, system availability increases if disks can be detached from a failed host and attached to a functioning one. In our scheme, the physical disk **always** contains all the information needed to manage it as a *DSF logical disk*, and thus it can easily be removed from one host and attached to another one.

3.8. Reads and Caching

Figure 3 illustrates how DSF reads a data block given a file name and an offset within that file.

To open a file, the Client first reads the file's parent directory block (labeled 1 in the diagram) to determine its inode address. Note that the parent directory is, itself, a data file that must be read using the procedure described above here. DSF breaks this recursion at the root; the Client learns the inode address of the root when it mounts the file system.

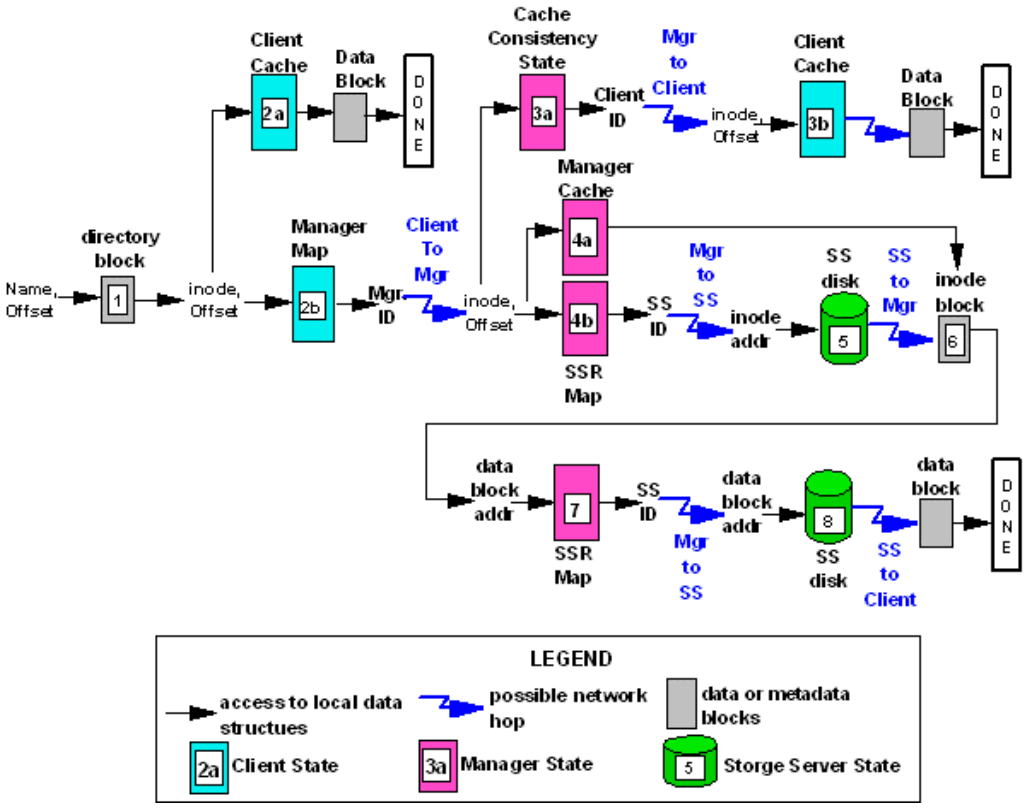

Figure 3 Read a Data Block

Once the Client determines the file inode address, it follows a Manager selection procedure, to locate/assign appropriate Manager for the file.

As the top left of the path in the figure indicates, the Client first checks in its local cache for the block (2a); if the block is present, the request is satisfied from the local cache. Otherwise, it follows the lower path to fetch the data block over the network. The Client first uses its manager map to locate the correct manager from the inode address (2b) then sends a Read request to the Manager. If the Manager is not co-located with the Client, this message travels over the network.

The Manager then tries to satisfy the request by fetching the data from the cooperative cache, i.e. from some other Client's cache. The Manager checks the cache consistency state (3a), and, if possible, forwards the request to another Client caching the requested data block. The "source" Client reads the block from

its local cache (3b) and forwards the data directly to the "destination" Client (the one that originated the request). The Manager is notified on the block arrival to the "destination" Client and adds it to the list of Clients caching the block that the manager maintains.

If no Client can supply the data from its cache, the Manager routes the Read request to disk storage by examining the inode block. The Manager may find the inode block in its local cache (4a) or it may have to read the inode block from disk. If the Manager has to read the inode from disk, its uses the inode address and the SSR (Storage Server Map) map (4b) to implicitly determine which Storage Server to contact. The manager then requests the inode block from the Storage Server, who then reads the metadata block from its disk and sends it back to the Manager (5). Once the Manager receives the inode block it uses the inode (6) to identify the address of the requested data block (if the file is large, the Manager may have to read several levels of indirect blocks to fine the data block's address; to do this the Manager follows the same procedure in reading indirect blocks as in reading the inode block; this is not shown here).

The Manager uses the data block's address and the SSR map (7) to send the Read request to the appropriate Storage Server keeping the block. The Storage server reads the data block from its disk (8) and forwards the block directly to the Client that originated the Read request.

4. Experimental Results

All DSF components where initially implemented on NT/4 and then ported to Linux. The NT version was up and running in 1999 and a heterogeneous system was running in the lab during the summer of 2000.

A test bed for experimental measurements was implemented on a cluster of three NT/x86 machines... as depicted in Figure 4

The clients ran on two machines, the File Managers ran on two machines and the

Storage Manager ran on two machines.

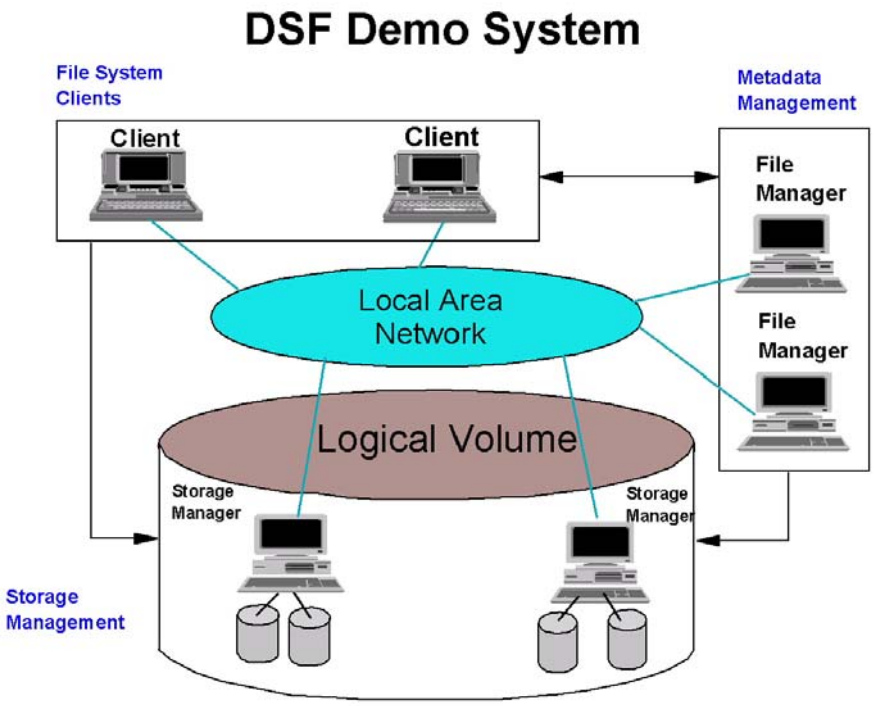

Figure 4 Measurement/Demo Systems

Test environment:

3 NT machines (one PC Pentium III, 660Mhz (client + manager), 2 PC Pentium II, 266Mhz - one running mangers and the second running client), 128 MB RAM on each PC. Each PC running NT 4 SP 5

Fast Ethernet (100 MB/sec) which is part of the site infrastructure (not a dedicated switch)

NT NTFS native cache: 21MB

DSF client - DSF managed cache: 6 MB/client (4k data blocks)

DSF File Manager - DSF managed cache: 2 MB/file manager (1k metadata blocks)

We did run two sets of tests:

- A BMP file read with on screen presentation using Internet Explorer (graphics presentation)

- A benchmark named Postmark from Network Appliance Inc. (Postmark)

4.1. Graphics presentation

Test file: a BMP file of size 4.8 MB (1172 blocks * 4kbytes /block)

Acronyms and special terms:

DSS - DSF storage Manager
FMGR - DSF file manager
DSF Client - A 2 layer NT driver (File system driver and Logical Volume & com driver)
CC - DSF cooperative cache mechanism

Test results per the test file that is read from storage and presented on the screen:

Table 1 File Access Experiment

DSF Client Remote DSS Access	DSF Client Local Cache File Access	DSF Client CC Access through FMGR	NT Native Local File Access	NT Native cache access
1.692 sec	8.79 msec	1.172 sec	1 sec	0.45 sec

Note:

DSF measurements were made with DSF internal diagnostic measurement tools

NT file access time measurements were done using a stop watch (the test program was a browser and NTFS is not instrumented).

As can be seen, on this experiment, CC file access time is 68 % of the DSS access time. Having a faster machine for the other DSF client machine will make the CC faster because CC operation is CPU bound

4.2. Postmark

Postmark v1.13 Experiment:

553 file creations and deletions and 100 file transactions (47 reads + 53 appends); read/write combinations determined by a coin toss. Total read 261.84 kB and total write 2.87 MB

Table 2 Postmark

	Total test time	File Create/sec	File Delete/sec	Read speed KB/sec	Write speed KB/Sec
NTFS	4	138	138	65.44	717.97
DSF(DSS local)	31	17	17	8.45	92.64
DSF(DSS remote)	33	16	16	7.93	87.03

The experiment results are displayed in Table 2. Please note that those results were obtained with "first cut code", not optimized and including modules where basic function was the only goal pursued.

5. Discussion

The DSF attempts to build a serverless storage access system that distributes all aspects of storage management over cooperating machines interconnected by a fast, switched network. The system should scale from two to several hundred machines, using commodity components (similar to the xFS goals).

DSF attempts to outperform xFS by using a dynamic (rather than xFS' static) management distribution policy.

DSF attempts to provide better reliability than xFS' by employing a simplified recovery mechanism, based on metadata shadowing (rather than xFS' log based mechanism), and by carrying out directory operations as atomic transactions at the storage level.

DSF is similar to Frangipani in using logical volumes to hide the distributed nature of the storage system from its clients. It outperforms Frangipani by employing dynamic file management, block level cache synchronization, and cooperative caching. DSF is designed to a higher level of scalability than the cluster based file systems: up to several hundreds of commodity workstations.

Cooperative caching: Like XFS, DSF clients are assumed to contribute main memory and CPU cycles to support the cooperative caching operations triggered by neighboring DSF clients.

Taken together, DSF has advantages over existing systems. It provides extensible distributed disk management, it moves functionality (e.g. allocation) down to lower levels (disk), and it provides stable storage and dynamic assignment of file managers.

6. Conclusions

We have presented the Data Sharing Facility system that is based on a novel design:

Use of logical volumes whereby logical addresses span several disks, self management of space, dynamic distribution of file management across multiple nodes (machines) for dynamic load balancing, cooperative caching, and stable and transactional storage in low layers (close to storage).

Since there are no dedicated machines, any machine may assume the responsibilities of a failed component, resulting in improved fault-tolerance.

Based on these design principles the DSF has the potential of improved features and properties in terms of scalability and recoverability over existing filesystems.

7. References

[1] "NFS Version 3 design and Implementation" Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel and David Hintz, Proceedings of summer USENIX Conference, pp. 137-152, June 1994.

[2] "Design and Implementation of the Sun Network File System" Russel Sandberg, David Goldberg, Steve Klieman, Dan Walsh, and Bob Lyon, Proceedings of summer USENIX Conference, pp. 119-130, June 1985.

[3] "Scale and Performance in Distributed File System" H.J. Kazar, M. Menees, S. Nichols, D. Satyanarayanan, M. Midebotham, ACM Trans. on Computer Systems, Vol 6, 1 Feb. 1988. pp. 51-81.

[4] "VAXclusters: A Closely-Coupled Distributed System" N. Kronenberg, H. Levy, and W. Strecker. ACM Transaction on Computer Systems, May 1986.

[5] "The Design and Implementation of a Distributed File System" Andrew C. Goldstein Digital Technical Journal, 1(5) pp. 45-55, September 1987.

[6] "Frangipani: A Scaleable Distributed File System" Chandramohan A. Tekkath, Timothy Mann, and Edward K. Lee, Digital Systems Research Center, 16th SOSP Conference.

[7] "Petal: Distributed Virtual Disks" E.K. Lee and C.A. Thekkath, ASPLOS, October 1996.

[8] "Serverless Network File Systems". T. Anderson, M. Dahlin, J. Neffe, D. Patterson, D. Roselli, and R. Wang. ACM Transaction on Computer Systems, February 1996.

[9] "Self-Stabilization" S. Dolev, the MIT Press, 208 pages, March 2000.

[10] "Communication Adaptive Self-Stabilizing Group Communication" Dolev S., and Schiller E., Technical Report #2000-02, Department of Mathematics and Computer Science Ben-Gurion University, Beer-Sheva, Israel, July 2000.

[11] "Efficient Cooperative Caching Using Hints" P. Sarkar and J. Hartman, USENIX Conference on Operating Systems Design and Implementation, October 1996.

[12] "Cooperative Caching: Using Remote Client Memory to Improve File System Performance" M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson. OSDI, November 1994.

[13] "NFS Version 4 Design Considerations" Sun Microsystems, Inc. June 1999 URL: http://www.landfield.com/rfcs/rfc2624.html after the caption. To update the tables of figures and tables by right clicking inside them and selecting Update All.

[14] Wiebren de Jonge, M. Frans Kaashoek and Wilson C. Hsieh. “The logical disk: a new approach to improve file systems”.

Proc. 14th Symp. on Operating Systems Principles, pages 15-28, Dec. 1989.