(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
22 May 2009 (22.05.2009)

PCT

(10) International Publication Number
WO 2009/064720 A2

(51) International Patent Classification:
G06F 17/30 (2006.01)

(21) International Application Number:
PCT/US2008/083117

(22) International Filing Date:
11 November 2008 (11.11.2008)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/987,206	12 November 2007 (12.11.2007)	US
60/988,306	15 November 2007 (15.11.2007)	US
60/988,269	15 November 2007 (15.11.2007)	US
60/987,181	12 November 2007 (12.11.2007)	US
60/987,197	12 November 2007 (12.11.2007)	US
60/987,174	12 November 2007 (12.11.2007)	US
60/987,194	12 November 2007 (12.11.2007)	US

(71) Applicant *(for all designated States except US)*: **ATTUNE SYSTEMS, INC.** [US/US]; 3255 Scott Boulevard Building 2, Santa Clara, CA 94054 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **SURESH, Suma** [IN/US]; 2985 Warburton Avenue, Santa Clara, CA 95051 (US). **MARINOV, Borislav** [BG/US]; 12 Mayflower Street, Aliso Viejo, CA 92656 (US). **MAKKAR, Chitra** [IN/US]; 3166 Rodney Common, Freemont, CA 94538 (US). **COIMBATORE, Saravanan** [CA/US]; 818 W. Remington Drive, Sunnyvale, CA 94087 (US). **VOGEL, Ron, S.** [US/US]; 1563 Chihong Drive, San Jose, CA 95131 (US). **MARINKOVIC, Vladan, Z.** [RS/US]; 21061 Dumetz Road, Woodland Hills, CA 91364 (US). **WONG, Thomas, K.** [US/US]; 1118 Mataro Ct, Pleasanton, CA 94566 (US).

(74) Agents: **MURPHY, Timothy, M.** et al.; Bromberg & Sunstein LLP, 125 Summer Street, Boston, MA 02110-1618 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE,

[Continued on next page]

(54) Title: LOAD SHARING, FILE MIGRATION, NETWORK CONFIGURATION, AND FILE DEDUPLICATION USING FILE VIRTUALIZATION

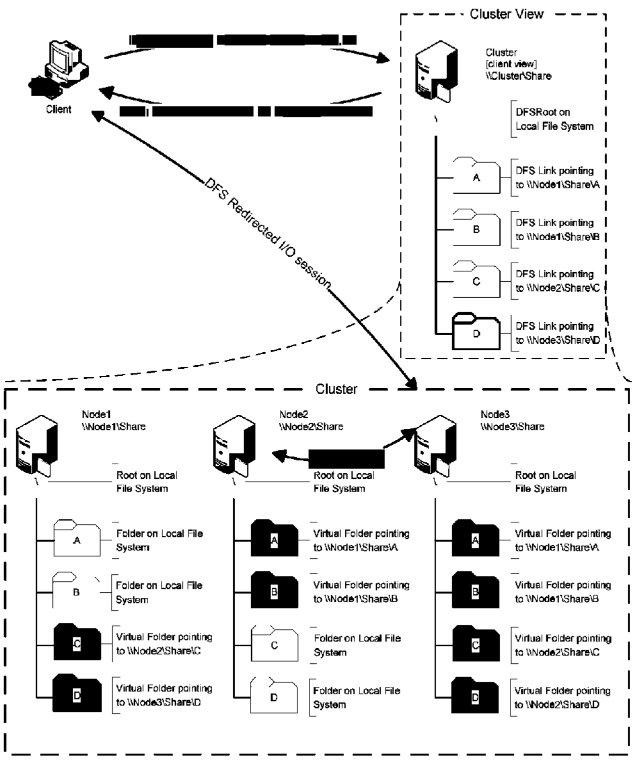

Cluster View
[redacted]
Cluster
[client view]
\\Cluster\Share
Client
[redacted]
DFSRoot on Local File System
DFS Link pointing to \\Node1\Share\A
DFS Link pointing to \\Node1\Share\B
DFS Link pointing to \\Node2\Share\C
DFS Link pointing to \\Node3\Share\D
DFS Redirected I/O session
Cluster
Node1
\\Node1\Share
Node2
\\Node2\Share
[redacted]
Node3
\\Node3\Share
Root on Local File System
Root on Local File System
Root on Local File System
A
Folder on Local File System
B
Folder on Local File System
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node3\Share\D
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Folder on Local File System
D
Folder on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node2\Share\D

FIG. A-11

(57) **Abstract:** File virtualization is used to provide for load sharing, file migration, network configuration, and file deduplication in storage networks.

WO 2009/064720 A2

EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

3193-132WO-966152
11/11/2008

LOAD SHARING, FILE MIGRATION, NETWORK CONFIGURATION, AND FILE DEDUPLICATION USING FILE VIRTUALIZATION

CROSS-REFERENCE TO RELATED APPLICATIONS

This PCT application claims priority from the following U.S. Provisional Patent Applications:

U.S. Provisional Patent Application No. 60/987,174 entitled **LOAD SHARING CLUSTER FILE SYSTEM** filed November 12, 2007 (Attorney Docket No. 3193/120);

U.S. Provisional Patent Application No. 60/987,197 entitled **HOTSPOT MITIGATION IN LOAD SHARING CLUSTER FILE SYSTEMS** filed November 12, 2007 (Attorney Docket No. 3193/122);

United States Provisional Patent Application No. 60/987,206 entitled **NON-DISRUPTIVE FILE MIGRATION** filed November 12, 2007 (Attorney Docket No. 3193/121);

United States Provisional Patent Application No. 60/987,194 entitled **ON DEMAND FILE VIRTUALIZATION FOR SERVER CONFIGURATION MANAGEMENT WITH LIMITED INTERRUPTION** filed November 12, 2007 (Attorney Docket No. 3193/123);

United States Provisional Patent Application No. 60/987,181 entitled **FILE DEDUPLICATION USING STORAGE TIERS** filed November 12, 2007 (Attorney Docket No. 3193/124);

U.S. Provisional Patent Application No. 60/988,269 entitled **FILE DEDUPLICATION USING COPY-ON-WRITE STORAGE TIERS** filed on November 15, 2007 (Attorney Docket No. 3193/125); and

U.S. Provisional Patent Application No. 60/988,306 entitled **FILE DEDUPLICATION USING A VIRTUAL COPY-ON-WRITE STORAGE TIER** filed on November 15, 2007 (Attorney Docket No. 3193/126).

This patent application also may be related to one or more of the following patent applications:

3193-132WO-966152
11/11/2008

United States Provisional Patent Application No. 60/923,765 entitled **NETWORK FILE MANAGEMENT SYSTEMS, APPARATUS, AND METHODS** filed on April 16, 2007 (Attorney Docket No. 3193/114).

United States Provisional Patent Application No. 60/940,104 entitled **REMOTE FILE VIRTUALIZATION** filed on May 25, 2007 (Attorney Docket No. 3193/116).

United States Provisional Patent Application No. 60/987,161 entitled **REMOTE FILE VIRTUALIZATION METADATA MIRRORING** filed November 12, 2007 (Attorney Docket No. 3193/117).

United States Provisional Patent Application No. 60/987,165 entitled **REMOTE FILE VIRTUALIZATION DATA MIRRORING** filed November 12, 2007 (Attorney Docket No. 3193/118).

United States Provisional Patent Application No. 60/987,170 entitled **REMOTE FILE VIRTUALIZATION WITH NO EDGE SERVERS** filed November 12, 2007 (Attorney Docket No. 3193/119).

United States Patent Application No. 12/104,197 entitled **FILE AGGREGATION IN A SWITCHED FILE SYSTEM** filed April 16, 2008 (Attorney Docket No. 3193/129).

United States Patent Application No. 12/103,989 entitled **FILE AGGREGATION IN A SWITCHED FILE SYSTEM** filed April 16, 2008 (Attorney Docket No. 3193/130).

United States Patent Application No. 12/126,129 entitled **REMOTE FILE VIRTUALIZATION IN A SWITCHED FILE SYSTEM** filed May 23, 2008 (Attorney Docket No. 3193/131).

All of the above-referenced patent applications are hereby incorporated herein by reference in their entireties.

FIELD OF THE INVENTION

The inventions described herein relate generally to storage networks and, more particularly, to load sharing, file migration, network configuration, and file deduplication using file virtualization in storage networks.

3193-132WO-966152
11/11/2008

BACKGROUND

In a “load balancing” cluster file system, different nodes in the cluster access the same portion or the entirety of the shared file system. Clients of the file system are either randomly connected to a node, or a group of clients are designated to connect to a specific node. Each node may receive a different load of client requests for file services. If a node is experiencing more requests than other nodes, the node may forward the request to a node with a lower load. Ideally, each node should get similar number of file requests from clients.

Because any node participating in the cluster can contain the authoritative state on any given file system object, every node can be a synchronization point for a file. Since two or more nodes may access the same file at the same time, complex distributed concurrency algorithms are needed to resolve any access conflict. These algorithms are hard to write and take years to become fully reliable to function properly in a production environment.

The GPFS file system developed by IBM is an example of a Load Balancing Cluster File System.

In a “load sharing” cluster file system, each cluster node is responsible for serving one or more non-overlapping portions of the cluster file system namespace. If a node receives client requests for data outside the scope of the namespace it is serving, it may forward the request to the node that does service the requested region of the namespace.

Since the server nodes do not share overlapped regions of the file system, only a single server will contain the authoritative state of the portion of the file system it serves, a single synchronization point exists. This removes the need for implementing complex distributed concurrency algorithms.

Load sharing cluster file systems generally provide such things as:

1) High Availability and Redundancy: Because the file system is configured within a cluster, cluster protection and availability are extended to the file system.

2) Reduced complexity: Since each node has exclusive ownership of the filesystem it servers, implementing a load sharing cluster filesystem becomes much simpler compared to a load balancing cluster file system where complex concurrency

algorithms are needed for arbitration of shared access by each node to the complete file system.

3) Increased Performance and Capacity: With the partitioning of the filesystem, additional nodes can contribute to the serving of the filesystem thus allowing higher total cluster capacity to serve the clients as well as improved performance under high load. Ability to partition namespace based on need: Allows end users to hierarchically structure data to match the representation of their business needs.

4) Pay as you go horizontal scaling: Namespace partitioning allows capacity and performance to expanded as there is need and in the area where the need is greatest, rather than globally in the cluster.

5) Enable real-time reconfiguration of the namespace: Unlike technologies like DFS where there is no ability to transparently reconfigure the namespace or contact all clients using the namespace, Load Sharing Cluster File Systems maintain statefull information about all connections and are able to provide seamless namespace reconfiguration via server side File Virtualization technology, solving one of the biggest deployment hurdles for technologies like DFS.

Since clients do not know how the cluster namespace is partitioned among the nodes of a cluster file system, the node that exports the entire namespace of the cluster file system will bear the full burden and will get all of the request traffic for the cluster file system. That node must then direct each request to the node that is responsible for the partitioned namespace. This extra hop adds additional latency and introduces a scalability problem. Furthermore, the workload of a Load Sharing Cluster is distributed among the nodes based on how the cluster namespace is partitioned. Certain namespaces may experience more workload than others, creating hotspots in the cluster file system. However, since only one node, the node that owns the partitioned namespace, is allowed to service requests for the partitioned namespace that it is responsible for; other nodes with low workload are not capable of helping nodes that are busy. Finally, reconfiguring the partitioned namespaces among the node usually involves moving data or metadata from one node to another and this data movement is very disruptive. Thus, while it is desirable to provide a load sharing cluster file system, these problems must be resolved first before a Load Sharing Cluster File System becomes practical.

Microsoft DFS allows administrators to create a virtual folder consisting of a group of shared folders located on different servers by transparently connecting them to one or more DFS namespaces. A DFS namespace is a virtual view of shared folders in an organization. Each virtual folder in a DFS namespace may be a DFS link that specifies a file server that is responsible for the namespace identified by the virtual folder, or it may be another virtual folder containing other DFS links and virtual folders. Under DFS, a file server that exports shared folders may be a member of many DFS namespaces. Each server in a DFS namespace is not aware that the file server is a member of a DFS namespace. In essence, DFS creates a loosely coupled distributed file system consisting of one or more file servers that operate independently of each other in the namespace.

DFS uses a client-side name resolution scheme to locate the file server that is destined to process file request for a virtual folder in a DFS namespace. The server that exports the DFS namespace in a root virtual folder, the DFS root server, will receive all the name resolution request traffic destined for the virtual folder.

The clients of a DFS namespace will ask the DFS root server who is the target file server and the shared folder in the file server that corresponds to a DFS virtual folder. Upon receiving the information, the DFS clients is responsible to redirect file requests to the target file server and a new path name constructed from the information obtained from the DFS root server. To reduce the load of the DFS root server, the DFS root server does not keep track of who are the clients of the exported DFS namespace. To further reduce the load, clients keep a cache of the association of a virtual folder and its target server and the actual pathname in the target server. Once the client processes the client-side resolution and connects to the target file server, the DFS server no longer participates in the network I/O. Furthermore, if a name is in the client side DFS cache, the client will not contact the DFS root server again for the same name until the cache is stale, usually for about 15 minutes. This methodology allows DFS great efficiency and optimal performance since the client is rapidly connected directly to the target file server.

Due to the client-side nature of the protocol and the fact that a connection is not maintained from the client to the DFS server, configuration changes in the DFS namespace cannot be propagated to the clients, especially since DFS does not maintain a client list. Further complicating the problem, each client also uses a client-side DFS name cache and it will not obtain the up-to-date file location information from the server unless the cache is stale. Therefore, maintaining configuration inconsistency is a big challenge. If configuration consistency is not maintained, even for a small duration, may lead to data corruption. Thus for DFS to become a viable cluster solution, configuration consistency must be maintained at all time.

Generally speaking, “file virtualization” is a method for a computer node to proxy client filesystem requests to a secondary storage server that has been virtually represented in the local portion of the file system namespace as a mounted folder.

A traditional file system manages the storage space by providing a hierarchical namespace. The hierarchical namespace starts from the root directory, which contains files and subdirectories. Each directory may also contain files and subdirectories identifying other files or subdirectories. Data is stored in files. Every file and directory is identified by a name. The full name of a file or directory is constructed by concatenating the name of the root directory and the names of each subdirectory that finally leads to the subdirectory containing the identified file or directory, together with the name of the file or the directory.

The full name of a file thus carries with it two pieces of information: (1) the identification of the file and (2) the physical storage location where the file is stored. If the physical storage location of a file is changed (for example, moved from one partition mounted on a system to another), the identification of the file changes as well.

For ease of management, as well as for a variety of other reasons, the administrator would like to control the physical storage location of a file. For example, important files might be stored on expensive, high-performance file servers, while less important files could be stored on less expensive and less capable file servers.

Unfortunately, moving files from one server to another usually changes the full name of the files and thus, their identification, as well. This is usually a very disruptive process, since after the move users may not be able to remember the new location of their files. Thus, it is desirable to separate the physical storage location of a file from its identification. With this separation, IT and system administrators will be able to control the physical storage location of a file while preserving what the user perceives as the location of the file (and thus its identity).

File virtualization is a technology that separates the full name of a file from its physical storage location. File virtualization is usually implemented as a hardware appliance that is located in the data path between users and the file servers. For users, a file virtualization appliance appears as a file server that exports the namespace of a file system. From the file servers' perspective, the file virtualization appliance appears as just a normal user. Attune System's Maestro File Manager (MFM) is an example of a file virtualization appliance. FIG. A-1 is a schematic diagram showing an exemplary switched file system including a file switch (MFM).

As a result of separating the full name of a file from the file's physical storage location, file virtualization provides the following capabilities:

1) Creation of a synthetic namespace

   Once a file is virtualized, the full filename does not provide any information about where the file is actually stored. This leads to the creation of synthetic directories where the files in a single synthetic directory may be stored on different file servers. A synthetic namespace can also be created where the directories in the synthetic namespace may contain files or directories from a number of different file servers. Thus, file virtualization allows the creation of a single global namespace from a number of cooperating file servers. The synthetic namespace is not restricted to be from one file server, or one file system.

2) Allows having many full filenames to refer to a single file

   As a consequence of separating a file's name from the file's storage location, file virtualization also allows multiple full filenames to refer to a single file. This is important as it allows existing users to use the old filename while allowing new users to use a new name to access the same file.

3) Allows having one full name to refer to many files

   Another consequence of separating a file's name from the file's storage location is that one filename may refer to many files. Files that are identified by a single filename need not contain identical contents. If the files do contain identical contents, then one file is usually designated as the authoritative copy, while the other copies are called the mirror copies. Mirror copies increase the availability of the authoritative copy, since even if the file server containing the authoritative copy of a file is down, one of the mirror copies may be designated as a new authoritative copy and normal file access can then resumed. On the other hand, the contents of a file identified by a single name may change according to the identity of the user who wants to access the file.

Cluster file systems may be used to meet strong growth of end user unstructured data needs. Load sharing cluster file system is generally simpler to implement than load balancing cluster file system. Furthermore, a cluster file system that uses partitioned namespace to divide workload among the nodes in a cluster is a better match for the business environment. This is because each organization in a business environment usually has its own designated namespace. For example, engineering department may own the namespace /global/engineering, while the marketing department owns /global/marketing namespace. If engineering needs more resources, engineering namespace may be further partitioned and more nodes are added for engineering, without affecting the marketing department.

DFS is good match for a load sharing namespace. Unfortunately, it is hard to maintain configuration consistency among all clients. It also is not a true cluster and does not provide protection from failure.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

Various embodiments of the present invention relate to load sharing, file migration, network configuration, and file deduplication using file virtualization in storage networks. Section A relates to load sharing cluster file systems. Section B relates to non-disruptive file migration. Section C relates to on demand file virtualization for server configuration management with limited interruption. Section D relates to file deduplication using storage tiers. Section E relates to file deduplication using copy-on-write storage tiers.

SECTION A - LOAD SHARING CLUSTER FILE SYSTEMS

Embodiments of the present invention relate generally to load sharing clusters in which each node is responsible for one or more non-overlapping subset(s) of the cluster namespace and will process only those requests that access file or directory objects in the partitioned namespace that the node controls while redirecting requests designated for other nodes. Specific embodiments of the present invention are based on using DFS in conjunction with File Virtualization to overcome DFS configuration consistency deficiency as well as to provide cluster protection and availability. Exemplary embodiments use DFS to enable clients to communicate directly with the node in the load sharing cluster that is destined to process the request according to the partitioned namespace that the request is for. Once the namespace for the node is resolved, DFS is essentially out of the picture. DFS resolution is essentially used as a hint. If the DFS configuration is changed and a node receives a request not destined for the node, the node will forward the request to the correct owner, thus overcoming the DFS configuration consistency problem.

In accordance with one embodiment of the present invention there is provides a method for load sharing in an aggregated file system having a cluster of file storage nodes and a distributed filesystem server (DFS) node, the file storage nodes collectively maintaining a shared storage including a plurality of non-overlapping portions, each file storage node owning at least one of the non-overlapping portions and including for each non-overlapping portion not owned by the file storage node a file virtualization link identifying another file storage node for the non-overlapping portion, the DFS node mapping each non-overlapping portion to a file storage node. The method involves generating client requests by a number of client nodes, each client request identifying a non-overlapping portion and directed to a specific file storage node based on an access to the DFS server or information in a client cache; and for each client request received by a file storage node, servicing the client request by the receiving file storage node if the receiving file storage node owns the identified non-overlapping portion and otherwise forwarding the client request by the receiving file storage node to another file storage node identified using the file virtualization links.

In various alternative embodiments, the method may further involve migrating a specified non-overlapping portion from a source file storage node to a destination file server node, for example, due to reconfiguration of the cluster of based on loading of the source file storage node. Migrating the specified non-overlapping portion may involve establishing a file virtualization link on the destination file server node, the file virtualization link identifying the file storage node that owns the non-overlapping portion; updating the cluster resource to map the non-overlapping portion to the destination file storage node; building metadata for the non-overlapping portion on the destination file storage node using sparse files such that all file and directory attributes of the non-overlapping portion are replicated on the destination file storage node without any data, and during such building, forwarding client requests received for the non-overlapping portion by the destination file storage node to the file storage node that owns the non-overlapping portion based on the file virtualization link; after building the metadata for the non-overlapping portion on the destination file storage, copying data for the non-overlapping portion from the source file storage node to the destination file storage node, and during such copying, servicing metadata requests received for the non-overlapping portion by the destination file storage node using the metadata and forwarding data requests received for the non-overlapping portion by the destination file storage node to the file storage node that owns the non-overlapping portion based on the file virtualization link; and after completion of the copying, designating the destination file storage node as the owner of the non-overlapping portion and thereafter servicing client requests received for the non-overlapping portion by the destination file storage node. The destination file storage node may be an existing file storage node in the cluster or may be a new file storage node added to the cluster.

In accordance with another embodiment of the present invention there is provided a method for load sharing by a file storage node in an aggregated file system having a plurality of file storage nodes and a distributed filesystem server (DFS) node, the file storage nodes collectively maintaining a shared storage including a plurality of non-overlapping portions, each file storage node owning at least one of the non-overlapping portions and including for each non-overlapping portion not owned by the file storage node a file virtualization link identifying another file storage node for the non-overlapping portion, the DFS node mapping each non-overlapping portion to a file storage node. The method involves receiving, by the file storage node, a client request identifying a non-overlapping portion; when the file storage node owns the identified non-overlapping portion, servicing the client request by the file storage node; and when the file storage node does not own the identified non-overlapping portion, forwarding the client request by the file storage node to another file storage node identified using the file virtualization links

In various alternative embodiments, the method may further involve migrating a specified non-overlapping portion from another file storage node. Migrating the specified non-overlapping portion may involve maintaining a file virtualization link to the specified non-overlapping portion on the other file storage node; migrating metadata for the specified non-overlapping portion from the other file storage node; after migrating the metadata, migrating data for the specified non-overlapping portion from the other file storage node; and after migrating the data, breaking the file virtualization link. While migrating the metadata, the file storage node typically redirects requests for the specified non-overlapping portion to the other file storage node. While migrating the data, the file storage node typically services metadata requests for the specified non-overlapping portion from the migrated metadata and forwards data request for the specified non-overlapping portion to the other file storage node. After breaking the file virtualization link, the file storage node typically services requests for the specified non-overlapping portion from the migrated metadata and data. Migrating may be done for at least one of load sharing and hotspot mitigation.

In accordance with another embodiment of the present invention there is provided a file storage node for use in an aggregated filesystem having a plurality of file storage nodes and a cluster resource, the file storage nodes collectively maintaining a shared storage including a plurality of non-overlapping portions, each file storage node owning at least one of the non-overlapping portions and including for each non-overlapping portion owned by another file storage node a file virtualization link identifying the other file storage node, the cluster resource including for each non-overlapping portion a link mapping the non-overlapping portion to a target file storage node. The file storage node includes a network interface for receiving a client request identifying a non-overlapping portion; and a processor configured to service the client request if the file storage node owns the identified non-overlapping portion and to forward the client request to another file storage node identified using the file virtualization links if the file storage node does not own the identified non-overlapping portion.

In various alternative embodiments, the processor may be further configured to migrate a specified non-overlapping portion from another file storage node. Migrating the specified non-overlapping portion may involve maintaining a file virtualization link to the specified non-overlapping portion on the other file storage node; migrating metadata for the specified non-overlapping portion from the other file storage node; after migrating the metadata, migrating data for the specified non-overlapping portion from the other file storage node; and after migrating the data, breaking the file virtualization link. While migrating the metadata, the file storage node typically redirects requests for the specified non-overlapping portion to the other file storage node. While migrating the data, the file storage node typically services metadata requests for the specified non-overlapping portion from the migrated metadata and forwards data request for the specified non-overlapping portion to the other file storage node. After breaking the file virtualization link, the file storage node typically services requests for the specified non-overlapping portion from the migrated metadata and data. Migrating may be done for at least one of load sharing and hotspot mitigation.

The foregoing features of the invention will be more readily understood by reference to the following detailed description, taken with reference to the accompanying drawings, in which:

FIG. A-1 is a schematic diagram showing an exemplary switched file system including a file switch (MFM) as known in the art;

FIG. A-2 is a schematic diagram showing a cluster with shared storage as known in the art;

FIG. A-3 is a schematic diagram showing a file virtualization based aggregated file system as known in the art;

FIG. A-4 is a schematic diagram showing a clustered DFS namespace as known in the art;

FIG. A-5 is a schematic diagram showing a load sharing cluster file system in accordance with an exemplary embodiment of the present invention;

FIG. A-6 is a schematic diagram showing client interaction with a load sharing cluster file system in accordance with an exemplary embodiment of the present invention;

FIG. A-7 is a schematic diagram showing direct client access with forwarding of the request using file virtualization, in accordance with an exemplary embodiment of the present invention;

FIG. A-8 is a schematic diagram showing a situation in which file virtualization is used to forward requests that are misdirected due to stale cache information, in accordance with an exemplary embodiment of the present invention;

FIG. A-9 is a schematic diagram showing a load sharing cluster file system in accordance with another exemplary embodiment of the present invention;

FIG. A-10 is a schematic diagram showing DFS redirection to an available node in accordance with another exemplary embodiment of the present invention;

FIG. A-11 is a schematic diagram showing client I/O redirection with file virtualization in accordance with another exemplary embodiment of the present invention;

FIG. A-12 is a schematic diagram showing metadata migration in accordance with another exemplary embodiment of the present invention;

FIG. A-13 is a schematic diagram showing data migration in accordance with another exemplary embodiment of the present invention; and

FIG. A-14 is a schematic diagram showing migration completion in accordance with another exemplary embodiment of the present invention.

Definitions. As used in this section and the accompanying claims, the following terms shall have the meanings indicated, unless the context otherwise requires.

A “cluster” is a group of networked computer servers that all work together to provide high performance services to their client computers.

A “node,” “computer node” or “cluster node” is a server computer system that is participating in providing cluster services within a cluster.

A “cluster file system” is a distributed file system that is not a single server with a set of clients, but instead a cluster of servers that all work together to provide high performance file services to their clients. To the clients, the cluster is transparent - it is just "the file system", but the file system software deals with distributing requests to elements of the storage cluster.

An “active-active file system cluster” is a group of network connected computers in which each computer (node) participates in serving a cluster file system.

Embodiments of the present invention relate generally to load sharing clusters in which each node is responsible for one or more non-overlapping subset(s) of the cluster namespace and will process only those requests that access file or directory objects in the partitioned namespace that the node controls while redirecting requests designated for other nodes. Specific embodiments of the present invention are based on using DFS in conjunction with File Virtualization to overcome DFS configuration consistency deficiency as well as to provide cluster protection and availability. Exemplary embodiments use DFS to enable clients to communicate directly with the node in the load sharing cluster that is destined to process the request according to the partitioned namespace that the request is for. Once the namespace for the node is resolved, DFS is essentially out of the picture. DFS resolution is essentially used as a hint. If the DFS configuration is changed and a node receives a request not destined for the node, the node will forward the request to the correct owner, thus overcoming the DFS configuration consistency problem.

In a standard shared storage cluster system, each computer node participating in the cluster is the owner of one or more non-overlapped regions of the shared storage. The storage is located on a shared bus such that any node in the cluster can access any storage region, as needed for maintaining cluster file system availability. Each non-overlapped storage region contains a hierarchical filesystem containing a root and a shared folder. The folder is shared using the SMB protocol. FIG. A-2 is a schematic diagram showing a cluster with shared storage.

File Virtualization is used on each cluster node to create a uniform representation of the aggregated filesystem such that each local filesystem contains a folder and file virtualization link to the filesystem of every non-overlapped storage region served by the remaining nodes. FIG. A-3 is a schematic diagram showing a file virtualization based aggregated file system.

A DFS Namespace is created as a Cluster Resource and for every region of the non-overlapped shared storage, a DFS link is created in the DFSRoot. The DFS namespace is shared using the SMB protocol. FIG. A-4 is a schematic diagram showing a clustered DFS namespace.

FIG. A-5 is a schematic diagram showing a load sharing cluster file system in accordance with an exemplary embodiment of the present invention. Here, each file server owns one or more non-overlapping portions (e.g., folders) of the aggregated filesystem and includes file virtualization links to the folders owned by other file servers. Specifically, Node1 owns folder A and includes file virtualization links to folder B in

Node2 and to folder X in NodeX, Node2 owns folder B and includes file virtualization links for folder A in Node1 and to folder X in NodeX, and Node X owns folder X and includes file virtualization links to folder A in Node1 and to folder B in Node2.

Note that each node share and the cluster share represent a uniform view of the cluster file system and that I/O can be satisfied at every entry point of the file system.

FIG. A-6 is a schematic diagram showing client interaction with a load sharing cluster file system in accordance with an exemplary embodiment of the present invention. Here, the client first sends an I/O request to the cluster resource (e.g., DFS server) including a file pathname (\\Cluster\Share\B\file.txt in this example). The cluster resource maps the file pathname to a file server that owns the file according to its own view of the aggregated filesystem (which may differ from the views maintained by one or more of the file servers for various reasons) and responds with a DFS reparse message that redirects the client to the file server selected by the cluster resource (the file pathname maps to Node2 in this example). The client updates its local MUP Cache to redirect all I/O destined for that particular pathname (i.e., \\Cluster\Share\B) to the specified location (i.e., \\Node2\Share\B) and then performs I/O directly to \\Node2\Share\B.

Should the client access the cluster node directly and request I/O to a portion of the file system that is not locally owned by that node, file virtualization will proxy (i.e., forward or redirect) the I/O request to the correct cluster node based on the file virtualization links maintained by the node. FIG. A-7 is a schematic diagram showing direct client access with forwarding of the request using file virtualization, in accordance with an exemplary embodiment of the present invention.

If an administrator alters the location of the stored data and the client has stale entries in its DFS MUP Cache, file virtualization will perform the necessary I/O proxy on the server. FIG. A-8 is a schematic diagram showing a situation in which file virtualization is used to forward requests that are misdirected due to stale cache information, in accordance with an exemplary embodiment of the present invention. Here, folder B has been moved to Node1 but the client continues to direct requests for the folder to Node2 based on its cached information. Node2 proxies the requests to Node1 using file virtualization.

Thus, clients consult the DFS node to identify the target file storage node that owns an unknown file object (e.g., a file object that has never been accessed before by the client). Once the file object is known to a client, the client sends file accesses to the file object directly to the identified target file storage node. The client may choose not to consult the DFS node again to identify the target node of the known file object until the client deemed it is necessary to consult the DFS node for the known file object again (e.g., when the cache entry expires). Over time, the information in the DFS node, the client caches, and/or the file storage nodes may become mismatched. As a result, a client may send a request to a file storage node (e.g., the node that the client thinks still owns the file object) that does not own the file object. Therefore, in embodiments of the present invention, the file storage nodes employ file virtualization techniques to direct misdirected file requests to the proper file storage node. It should be noted that it is possible for the view of the global namespace maintained by a particular file storage node to be incorrect and so one file storage node may misdirect the request to another file storage node that does not own the file object, but each file storage node will use file virtualization to forward the request as appropriate.

In a load sharing cluster filesystems of the type described above, each participating node of the cluster owns exclusive access to a non-overlapped portion of the shared file system namespace. If a node is experiencing a high number of client requests, it generally cannot distribute any portion of those requests to other nodes in the cluster. This may cause hotspots in the files system, where certain portions of the namespace experience high client request volume. If the high volume of requests causes the node to reach its performance limits, clients may experience degraded performance.

This hotspot problem may be mitigated, for example, by moving a portion of the aggregated filesystem from one node to another and/or by repartitioning or dividing the original namespace that experiences the hotspot problem into one or more smaller, non-overlapping sub-namespaces. Additional new nodes may be added to the cluster, or existing under-utilized nodes may be designated to take over the newly created namespaces. Before the reconfiguration of the namespace is complete, the metadata and data must be migrated from the old node to the newly created or existing newly responsible nodes.

Traditionally, moving data to a new server is very time consuming and disruptive to the client. During the migration, the data to be migrated is usually taken off-line and therefore data availability is generally reduced during the migration. Also, clients of the cluster file system must know the new configuration and the new pathname to the cluster file system changes, causing difficulties for the client.

Thus, certain embodiments of the present invention use file virtualization and DFS redirection as discussed above in combination with a non-disruptive server-side data mirroring/migration technique to permit portions of the aggregated filesystem to be moved among the nodes. These embodiments generally maintain data availability during the migration, so clients in general will not be aware of the server side operations. Since the migration is performed behind the cluster, the path to the data does not change. Thus, by combining the three technologies, a load sharing cluster file system is created that supports non-disruptive configuration changes. Furthermore, the clients of this cluster are able to talk to the correct node that is destined to handle their requests, bypassing most of the need for server-side namespace switching and concurrency control, resulting in a very efficient and scalable cluster file system.

As discussed above, in a "load sharing" cluster, each cluster node is responsible for serving one or more non-overlapping portions of the cluster file system namespace. If a node receives client requests for data outside the scope of the namespace it is serving, it may forward the request to the node that does service the requested region of the namespace.

Each portion of the namespace that a node exclusively serves to clients is an actual folder in the nodes local file system. Portions of the namespace served by other nodes will appear as a virtual folder, internally identifying the file server and the full path name where the virtual folder is located via File Virtualization methods.

For example, in FIG. A-9, node 1 is responsible for the namespace \\Cluster\Share\A, and \\Cluster\Share\B. They appear as real folders \A and \B in node 1 respectively. On the other hand, node 1 is not responsible for the namespaces \\Cluster\Share\C and \\Cluster\Share\D. These namespaces appear as virtual folders \C and \D on node 1 respectively. Normally, node 1 only receives requests targeted for \A and \B. However, if there is an inadvertent request directing at the virtual folder \C or \D, possibly because of a delay in propagating a namespace configuration changes, the request will be redirected by node 1 to the node that is responsible for \C (node 2) and \D (node 3) respectively.

Now say, for example, the cluster namespace is to be reconfigured so that node 2 is no longer responsible for the partitioned namespace \\Cluster\Share\D. Instead, node 3 will be the node to take on the additional responsibility of managing \\Cluster\Share\D. In order for this change to be effective, the local filesystem folder D on Node2 will need to be migrated to Node3. For the time being, though, Node3 will have a file virtualization link to the folder on Node2, as shown in FIG. A-10.

In an exemplary embodiment, the first step is to use DFS to redirect client requests to the target node that will receive the migrated data. Even though Node3 will use File Virtualization to redirect the data requests to Node2, this will ensure that the namespace remains available during the migration of the data files later in the process. This is illustrated in FIG. A-11.

Over time, when the client DFS cache synchronizes with the DFSRoot (typically around 15 minutes), client I/O requests will all go through Node3 and utilize the server-side file virtualization link from Node3 to Node2.

The next step is to build metadata in the local file system folder on Node3 using sparse files such that all file and directory attributes of the original data in Node2 are replicated in Node3 without any of the data, as illustrated in FIG. A-12. In this process, the share D on Node2 becomes a "Native with Metadata" Virtualized folder on Node3. This allows Node3 to start serving all metadata requests from the local metadata, such as locking, date and time, user authentication etc. Data requests remain proxied to Node2.

Once the complete Metadata is created on Node3, the data can be mirrored from Node2 to Node3, as illustrated in FIG. A-13.

When Mirror is complete, the mirror between Node2 and Node3 is broken, as illustrated in FIG. A-14. Migration is now complete, and the client load that was completely on Node2 is now distributed between Node2 and Node3.

SECTION B – NON-DISRUPTIVE FILE MIGRATION

In a computer network, NAS (Network Attached Storage) file servers provide file services for clients connected in a computer network using networking protocols like

CIFS or any other stateful protocol (e.g., NFS-v4). Usually, when a file, directory, or a server share is migrated from one server to another, the administrator takes the server offline, copies the files to the destination server, and finally brings the destination server online. The larger the amount of data been migrated, the longer the clients must wait for the migration to complete, which leads to longer server down-time.

In today's information age of exponentially growing server capacity and clients spread all over the globe, the amount of down-time an administrator can afford is constantly shrinking. It becomes almost impossible to migrate files from one server to another. This forces storage administrators to buy servers with significantly greater capacity (i.e., overprovision) in order to avoid/delay the need of migrating server data to a newer, higher capacity model.

A common approach to migrate files is to start migrating files while the source server is continued to be accessed and gradually copy all files to the destination server. On the subsequent passes only the newly modified files and directories (since the last pass) are copied and so on. This process is repeated until all files are migrated to the destination server. At this point, the source server is taken offline and replaced with the destination server, thus lowering the amount of time needed to migrate from one server to another. Although this solution lowers the down time it does not completely solve the problem with files that are constantly accessed or held open in exclusive mode. For those files, the user still suffers a visible access interruption and will have to invalidate all of its open handles and suffer service interruption during the migration of those files.

File Virtualization is a very powerful server management tool that normally is used for mirroring and load balancing for virtualized systems. Native Volume with Metadata is the only known way to bring File Virtualization to places where preserving the user's native directory structure is a must. Using File mirroring over Native Volume with Metadata is an excellent way to provide non-disruptive migration for storage servers.

In accordance with one aspect of the invention there is provided a method and file switch for non-disruptive migration of a native mode volume from a source server to a destination server. Such non-disruptive migration involves converting, by the file switch, the source native volume to a native with metadata volume using a local file system managed by the file switch; converting, by the file switch, the native with metadata volume to a mirrored native with metadata volume including the source server and the destination server, the destination server including a mirror copy of the native with metadata volume; removing, by the file switch, the source server from the mirrored native with metadata volume; and converting, by the file switch, the mirror copy of the native with metadata volume on the destination server to a destination native volume on the destination server.

In various alternative embodiments, converting the source native volume to the native with metadata volume may involve for each source directory in the source native volume, creating a corresponding local directory in the local file system including metadata associated with the source directory copied from the source native volume; and for each source file in the source native volume, creating a corresponding local sparse file in the local file system including file attributes copied from the source native volume but excluding the file contents associated with the source file. The metadata associated with the source directory copied from the source native volume may include directory security descriptors. Creating a local directory for a source directory may involve opening the source directory in the source native volume; placing a lock on the source directory; and creating the local directory and its metadata. Converting the native with metadata volume to the mirrored native with metadata volume may involve for each local directory, creating a corresponding destination directory in the destination server and maintaining a mapping of the local directory to a source directory pathname for the corresponding source directory in the source server and to a destination directory pathname for the corresponding destination directory in the destination server; and for each local file, creating a corresponding destination file in the destination server including file data copied from the source native volume and maintaining a mapping of the local file to a source file pathname for the corresponding source file in the source server and to a destination file pathname for the corresponding destination file in the destination server. Each mapping may include an indicator of the number of servers associated with the mirrored native with metadata volume. Removing the source server from the mirrored native with metadata volume may involve disabling usage of the source destination pathnames and the source file pathnames. Converting the mirror copy of the native with metadata volume on the destination server to a destination native volume may involve replicating state information for the destination directories and the destination files from the source native volume; disabling usage of the local directories and local files; and advertising the destination directories and destination files as a native volume. Converting the mirror copy of the native with metadata volume on the destination server to a destination native volume further may involve deleting unneeded metadata associated with the mirror copy of the native with metadata volume from the destination server.

The foregoing and advantages of the invention will be appreciated more fully from the following further description thereof with reference to the accompanying drawings wherein:

FIG. B-1 is a schematic block diagram of a two server system demonstrating file access from multiple clients;

FIG. B-2 is a schematic block diagram of a two server system where one of the servers is taken off the grid for migration;

FIG. B-3 is a schematic block diagram of a two server system where one of the servers was replaced by the new server after all files were copied from the old one;

FIG. B-4 depicts the process sequence of server migration with minimal interruption;

FIG. B-5 depicts the process sequence of non-disruptive server migration;

FIG. B-6 is a practical example of a sample global namespace including the metadata information and how the global name-space is used to calculate the target path;

FIG. B-7 is a practical example of a sample global namespace including the metadata information and how the global name-space is used to calculate the target paths; and

FIG. B-8 is a logic flow diagram for non-disruptive file migration by a file switch in accordance with an exemplary embodiment of the present invention.

Definitions. As used in this section and related claims, the following terms shall have the meanings indicated, unless the context otherwise requires:

Aggregator. An "aggregator" is a file switch that performs the function of directory, data, or namespace aggregation of a client data file over a file array.

File Switch. A "file switch" is a device (or group of devices) that performs file aggregation, transaction aggregation, and directory aggregation functions, and is physically or logically positioned between a client and a set of file servers. To client devices, the file switch appears to be a file server having enormous storage capabilities and high throughput. To the file servers, the file switch appears to be a client. The file switch directs the storage of individual user files over multiple file servers, using mirroring to improve fault tolerance as well as throughput. The aggregation functions of the file switch are done in a manner that is transparent to client devices. The file switch preferably communicates with the clients and with the file servers using standard file protocols, such as CIFS or NFS. The file switch preferably provides full virtualization of the file system such that data can be moved without changing path names and preferably also allows expansion/contraction/replacement without affecting clients or changing pathnames. Attune System's Maestro File Manager (MFM), which is represented in FIG. B-5, is an example of a file switch.

Switched File System. A "switched file system" is defined as a network including one or more file switches and one or more file servers. The switched file system is a file system since it exposes files as a method for sharing disk storage. The switched file system is a network file system, since it provides network file system services through a network file protocol--the file switches act as network file servers and the group of file switches may appear to the client computers as a single file server.

Native File System. A "native file system" is defined as the native file system exposed by the back-end servers.

Native mode. A "native mode" of operation is a mode of operation where the backend file system is exposed to the clients through the file switch such that the file switch completely preserves the directory structure and other metadata of the back end server. Each file server (share) represents a single mount point in the global namespace exposed by the file switch.

File. A file is the main component of a file system. A file is a collection of information that is used by a computer. There are many different types of files that are used for many different purposes, mostly for storing vast amounts of data (i.e., database files, music files, MPEGs, videos). There are also types of files that contain applications and programs used by computer operators as well as specific file formats used by different applications. Files range in size from a few bytes to many gigabytes and may contain any type of data. Formally, a file is a called a stream of bytes (or a data stream) residing on a file system. A file is always referred to by its name within a file system.

User File. A "user file" is the file or file object that a client computer works with (e.g., read, write, etc.), and in some contexts may also be referred to as an "aggregated file." A user file may be mirrored and stored in multiple file servers and/or data files within a switched file system.

File/Directory Metadata. A "file/directory metadata," also referred to as the "the metadata," is a data structure that contains information about the position of a specific file or directory including, but not limited to, the position and placement of the file/directory mirrors and their rank. In embodiments of the present invention, ordinary clients are typically not permitted to directly read or write the content of "the metadata", the clients still have indirect access to ordinary directory information and other metadata, such as file layout information, file length, etc.. In fact, in embodiments of the invention, the existence of "the metadata" is transparent to the clients, who need not have any knowledge of "the metadata" and its storage.

Mirror. A "mirror" is a copy of a file. When a file is configured to have two mirrors, that means there are two copies of the file.

Oplock. An oplock, also called an "opportunistic lock" is a mechanism for allowing the data in a file to be cached, typically by the user (or client) of the file. Unlike a regular lock on a file, an oplock on behalf of a first client is automatically broken whenever a second client attempts to access the file in a manner inconsistent with the oplock obtained by the first client. Thus, an oplock does not actually provide exclusive access to a file; rather it provides a mechanism for detecting when access to a file changes from exclusive to shared, and for writing cached data back to the file (if necessary) before enabling shared access to the file.

This section relates generally to migrating file data from one storage server to another in a non-disruptive manner using a stateful network file protocol such as CIFS.

Regular Migration

FIGs. B-1, B-2, and B-3 demonstrate how the standard (non-optimized) file migration is done. FIG. B-1 is a schematic block diagram of network file system before the beginning of the migration. Client11 to Client1m are regular clients that connect to the two back-end servers (Server11 and Server12) through a regular IP switch over a standard network file system protocol CIFS and/or NFS. When the administrator takes the server offline, he connects it directly to the destination server and begins direct file copy from source (Server21) to the destination (Server23) as depicted in FIG. B-2. When all files are copied, the administrator renames the destination server to the name of the source server and finally the administrator connects the destination server in place of the source server as shown in FIG. B-3.

Migration with Minimal Interruption

FIG. B-4 depicts the minimal disruption migration. All accessible files are migrated from Server41 to Server43. Since the process can take a long time, some of the files may get changed during migration. In the second step, those files are migrated (again). Step two is repeated until all files are migrated or until the amount of data remaining to be migrated falls under a predetermined amount. Finally, the migration is completed in a way similar to the regular migration: in Step n+1 Server41 and Server43 are taken offline. In step n+2, the remaining files are copied to the destination. In the final step (n+3), the server is renamed to the name of the source server and the destination server is brought on-line (n+4).

Non-Disruptive Migration

For stateful file system protocols, there are two major obstacles for providing non-disruptive migration: files that are constantly been updated and files kept open continuously.

Generally speaking, when a file is constantly updated, the file migration is constantly going to be triggered. If the file is relatively large the migration process will have to start keeping track of the modified regions. Otherwise, the algorithm is never going to be able to catch up with the modifications.

If a file is held open, its sharing mode may not allow the file to be opened by the migration process which will prevent copying the file to the destination server.

Normally these limitations can only be overcome by taking the server down while these files are been migrated. For the duration of this migration, the clients suffer a disruption in their ability to access those files.

Embodiments of the present invention described below utilize file virtualization in order to provide non-disruptive file/server migration. As shown in FIG. B-8, non-disruptive file migration can be summarized in four general steps:

1) Convert the source server from a Native volume to a Native with metadata volume (block 802).

2) Convert the native with metadata volume to a mirrored native with metadata volume, where the second mirror resides on the destination server (block 804).

3) Convert back to a native with metadata volume by removing the source server from the volume (block 806).

4) Finally, the native volume with metadata is converted to a simple native volume (block 808).

Native Volume

A native volume is a basic virtualized representation of a share from the back-end server. Its content (directories and files) are completely managed by the hosting file server. Clients can access the virtualized volume through the global namespace or directly by accessing the back-end server.

Native Volume with Metadata

A native volume with metadata is a natural extension of the native volume mode with the ability to keep additional metadata information for each file/directory. “The metadata” will keep at least the following information: the number of mirrors and a list of the destinations where the file/directory mirror is placed.

One embodiment of this is where a local NTFS directory is used for storing all information about the native volume. In this case, the whole remote namespace (without the file data) is replicated inside this directory. All file attributes (including security, EA, file size, etc) are preserved on all mirrors as well as in the file switch namespace.

To calculate the actual path of a file, the system replaces the top level file prefix with the one specified in the metadata and leaves the rest of the path unchanged. This operation is very similar to the DFS/MUP operation. FIG. B-6 is a practical example of a sample global namespace including the metadata information and how the global name-space is used to calculate the target path.

Mirrored Native Volume with Metadata

"Mirrored Native Volume with Metadata" is similar to the "Native Volume with Metadata" except there are two or more copies of the data. For the purpose of this embodiment, only two copies are used. FIG. B-7 is a practical example of a sample global namespace including the metadata information and how the global name-space is used to calculate the target paths.

Basic Operations for (Mirrored) Native Volume with Metadata

CREATE NEW FILE/DIRECTORY - When create operation comes, the operation is performed initially over the file in the Local NTFS drive. If it succeeds, a file metadata is created as well and associated with the file/directory (e.g., stored inside an alternate data stream) and than the operation is forwarded to all mirrors in parallel. When all mirrors complete the operation, the operation is completed back to the client.

OPEN EXISTING FILE/DIRECTORY - When an open operation comes, the operation is performed initially over the local NTFS file. This allows the file security permissions to be evaluated locally and force evaluation of the sharing mode. If it succeeds, the metadata is read, to get the file placement and mirrors after which the open operation is forwarded simultaneously to all mirrors. When all mirrors complete the open, the open operation is completed back to the client.

READ/WRITE OPERATIONS - Data operations are submitted simultaneously to all mirrors with the operation sent to the mirrors in their rank order. When all of them complete the operation is acknowledged to the client. No read/write data is stored on the local disk so there is no need to send data operations to it. RANGE-LOCK OPERATIONS - Advisory range-locks or mandatory range-locks may be implemented.

If advisory range-locks are supported, than the range-lock requests are sent only to the local NTFS volume. For mandatory range-locks, the range-lock requests are sent to the local file and after it succeeds it is sent to all mirrors. In this case the local file acts as an arbiter for resolving range-lock conflicts and deadlocks.

OPPORTUNISTIC LOCK (OP-LOCK) OPERATIONS - Oplock operations are submitted to local file and all mirrors in parallel. When (any) oplock breaks, the original client request is completed, although nothing is completed if the oplock level was already lowered. To produce the correct result, an exemplary embodiment starts (initially) with an uninitialized level which is the highest oplock level. From there on, the oplock level can only go down. Please note that it is possible the oplock level on mirror 1 to be broken to level 2 and while we are processing it, the level can be broken to level 0 on mirror 2. If the user acknowledges the break to level 2, it is failed immediately without sending anything to the mirrors. It should be noted that oplock break operations are the only operations that treats status pending as an acknowledgement that the operation completed successfully (i.e., processing it in a work item or from a different thread is unacceptable).

DIRECTORY ENUMERATION - All directory operations are served by the local name space. Since the local directory is a copy of the native directory structure, everything that the client requires is stored there.

DELETE AND RENAME OPERATIONS - The delete/rename operations are sent to the local directory first and after it succeeds it is sent to all file/directory mirrors (in parallel). The operation is completed when all mirrors completes it.

DIRECTORY CHANGE NOTIFICATIONS - Directory operations are submitted to all mirrors. Pass back the response when it comes. If there is no request to be completed, MFM saves the responses in their arrival order. When a new dir-change-notification request comes, it will pick the first pending response and complete it to the client, the next one will pick the next pending and so on. It is possible for the client to receive more than one break notification for the same change – one for the local metadata and one for each of the mirrors. This behavior is acceptable since the directory notifications are advisory and not time sensitive. The worst that can happen is the client will have to reread the state of the affected files. If there is no pending completion, than we submit directory change notification request to all mirrors that have no pending directory notification.

Converting from Native Volume to Native with Metadata Volume

In order to convert the Native Volume to a Native with metadata, all access to the back end server that is being converted will go through the file switch, i.e., the file switch is an in-band device. There should be no file access that does not go through it. A data corruption is possible in case files are been modified/accessed not through the file switch. The file switch cannot not enforce that the access to the backend servers is done only through the file switch.

Conversion from native to extended native is done by walking down the source directory tree and converting the volume directory by directory. Each directory operation usually is run by a single execution thread.

The execution thread opens the source directory, places a batch oplock on the source directory, so it can be notified in case someone changes it. In case the batch oplock is broken, the thread re-adds directory to the end of the list of directories to be processed, releases any resources it has acquired and exits.

Then the corresponding local directory and its metadata are created. The directory is enumerated and for each of the files found a sparse file is created in the local file system. The sparse file size corresponds to the actual file size. All other file attributes (time, attributes, security descriptors and EAs) are copied as well. The creation of "the metadata" for the file completes the conversion of the file.

After file enumeration completes, all directories are enumerated and for each directory found a new work item is created. The work items are added to the list of directories to be converted as a batch when the enumeration is completed. This would ensure that the sub-directory conversion will start only after the parent directory conversion is completed and avoid any nasty concurrency problems. At some point later when the same directory is scheduled again, any files and/or directories that have already been converted (by the previous attempts) would be skipped. This approach, although slow, can guarantee that there would be no missed entities.

The directory oplock break status is checked after processing each directory entity (file and/or directory). The status of the oplock break is not checked during the batch adding of the sub-directories to the directory processing queue since this operation is entirely local and is executed almost instantaneously.

All security descriptors are copied verbatim (without looking into it) except for the top level directory. The root directory security descriptor is converted to effective security descriptor and than set in the local NTFS directory. This would allow the sub-entities to properly inherit their security attributes from their parents.

This process repeats until there are no more entries in the directory list. The number of simultaneously processed directories can be limited to a predefined number to avoid slowing the system down due to over-parallelism. While converting the volume, the in memory structures of the currently opened files and directories maintained by the file switch (FIG. B-5) needs to be modified to comply with the requirements of the native with metadata volume structure.

To provide atomicity, some operations may require a temporal suspension of all operations over the affected entity (file or directory). In this case the access to the file/directory is suspended, the system waits for all outstanding operations (except range-locks with timeout) to complete and than it performs the required operation. When the operation completes, with success or a failure, the access to the entity is restored.

Usually, the temporary access suspension is at most several hundreds of milliseconds long, which is comparable to the network latency, and thus would not affect the applications using those files even if they are actively using the opened file.

Operations during Conversion to Native Volume with Metadata

If the file/directory does not have metadata (i.e., it is not converted yet), the operation is forwarded to the native volume otherwise the operations are served way it is described in "**Basic Operations for (Mirrored) Native Volume with Metadata**" with the following exceptions.

CREATE NEW FILE/DIRECTORY - This operation is performed in the local namespace. If it succeeds, it is processed as described in "**Basic Operations for (Mirrored) Native Volume with Metadata.**" If it fails, the operation is submitted to the native volume and if it succeeds, this is an indication that the local directory has not been created/converted yet. It will be created eventually so there really is nothing to do here.

CONVERTING THE IN-MEMORY RANGE-LOCK STRUCTURES - The range-lock requests can be handled in one of two possible ways: as advisory locks or as mandatory locks (Windows default). If advisory range-locks are supported, access to the file is suspended temporarily, and all range-lock requests are submitted to the local NTFS volume on the File Switch after which all pending requests on the source file are cancelled. Once cancelled access to the file is restored. If mandatory range-locks are supported, access to the file is suspended, and all range-lock requests are submitted to local NTFS volume first, followed by the range-lock requests being submitted to the other file mirrors. After the range-locks are granted, access to the file is restored. While the migration is running, open file and/or directory requests should be submitted in parallel to the local NTFS file system metadata and to the native volume. If the request succeeds on the backend server but fails on the local volume, this is an indication that the file/directory has not been converted yet. In this case, all parent directories inside the Local NTFS volume need to be recreated before the operation is acknowledged to the client.

CONVERTING OPPORTUNISTIC LOCK (OP-LOCK) OPERATIONS - Converting opportunistic lock operations from Native to Native Volume with metadata involves submitting an oplock to the local NTFS volume in order to make it compliant with the expected model.

CONVERTING ACTIVE DIRECTORY ENUMERATION - Since directory operation is a relatively short operation, there really is nothing special that needs to be done here. The operation would be completed eventually and then served the proper way.

RENAME OPERATIONS - There are four different rename operation combinations based on the file conversion state and the destination directory conversion state: both are converted, both are not converted; only the source is converted, and only the destination is converted. Nothing special is needed if both are converted. If the source is converted but the destination directory does not exist in the local NTFS volume, the destination directory is created in the local volume and the rename/move operation is performed on the native volume and on the NTFS volume. If the destination directory is converted, but the local file is not, the file is converted after the rename operation completes. If the destination directory is converted, but the local directory is not, the directory name is added to the list of directories that require conversion. If the source and the destination are not converted, the rename operation is executed over the native volume only. After the operation completed, the destination directory is checked one more time and in case the destination directory suddenly becomes converted, and the entity is a file, metadata is created for it; if the entity is a directory, it is added to the list of directories that require conversion. This behavior is done to ensure that an entity conversion will not be missed.

CONVERTING DIRECTORY CHANGE NOTIFICATIONS - Converting the directory change notifications from Native to Native Volume with metadata involves submitting a directory change notification to the local NTFS volume in order to make it compliant with the expected model.

Creating/rebuilding data mirrors for Native mode with Metadata Volume

The directory operations and walking the tree is very similar to converting the volume to extended-native mode. For each directory found, a new destination directory is created and all directory attributes are copied there as well.

When the source file is opened for reading, a filter oplock is placed on the local NTFS file (filter oplocks are not supported across the network). If this filter oplock gets broken because someone opened the file, the mirroring process is stopped, the uncompleted mirrors are deleted, and the file is put on a list for later attempts to mirror.

If a file/directory open fails with a sharing violation error, this file/directory is added to list to be processed at some time later when the file is closed or opened with more appropriate sharing mode.

Periodically the list of files with postponed mirroring is checked and the mirroring attempt is repeated.

After several unsuccessful attempts to mirror file data, an open file mirroring is performed. The process starts by creating an empty file where the new mirrors are placed and begins to copy file data. The file data is read sequentially from the beginning of the file until the end of the file and is written to all of the mirrors (please note that no file size increase is allowed during this phase). In addition, all client write (and file size change) requests are replicated and sent to all mirrors. To avoid data corruption, reading the data from the source and writing it to the mirror(s) is performed while user access to this file is suspended. The suspension is once again performed for a relatively small interval so as not be noticed by the user (or application).

When the file is mirrored, the file handle state is propagated to the new mirror as well. This state includes but is not limited to: mirror file handle, range-locks and oplocks. Range-locks are replicated to all mirrors only if mandatory range-locks are supported; otherwise, there is nothing more that needs to be done if only advisory locks are supported.

When a directory is mirrored, any directory change notifications request needs to be resubmitted to the new mirror as well.

Removing the Source Server from the Mirrored Volume

Convert back to a native with metadata volume is done atomically by programmatically setting the source server state to "force-removed", changing a global state to removing a mirror and logging off from the server. All operations pending on this server would be completed by the backend server and the file switch will silently "eat" them without sending any of them to the client.

After this, the source server references can be removed from "the metadata": the directory operations and walking the tree is very similar to the way the data mirrors are rebuild described at "**Creating/rebuilding data mirrors for Native mode with Metadata Volume**". Only the metadata structure is updated by removing the source server references from "the metadata". Finally, the in-memory data handle structures are updated to remove any references to the source server. All those operations can be performed with no client and/or application disruption.

Converting from Native with Metadata to a Native Volume

Converting starts by going through all currently opened handles and replicating the opened state (e.g. range locks directory notifications, oplocks, etc.) over the native volume.

When done, ALL access to the specified server set is temporarily suspended and all open files/directories on the local NTFS directory are closed (any operations failed/completed due to the close are ignored). The global state of the volume is set to a pure native volume so all new open/creates should go to the native volume only.

Finally, access to the volume is restored.

At this point, the metadata directory can be moved to a separate NTFS directory where all files and directories containing "the metadata" can be deleted and associated resources can be freed.

All those operations are performed with no client and/or application disruption.

SECTION C – LOAD SHARING FILE SYSTEM CLUSTERS

In a computer network, NAS (Network Attached Storage) file servers provide file services for clients connected in a computer network using networking protocols like CIFS or any other stateful protocol (e.g., NFS-v4). Many companies utilize various file Virtualization Appliances to provide better storage utilization and/or load balancing. Those devices usually sit in the data path (in-band) between the clients and the servers and present a unified view of the name spaces provided by the back-end server. From the client perspective, this device looks like a single storage server; for the back-end servers, the device looks like a super client that runs a multitude of users. Since the clients cannot see the back-end servers, the virtualization device is free to move, replicate, and even take offline any of the user's data, thus providing the user with a better user experience.

Earlier attempts at storage virtualization includes Microsoft Distributed File System (DFS) for presenting a single namespace, but these solutions are out-of band solutions where the client machine directly accesses the back-end servers but hides this from its users and applications. Out of band solutions have the benefit of being extremely fast, but unfortunately do not allow easy and seamless migration and or load balancing between different back-end servers.

In-line file virtualization is the next big thing in Storage but it does come with some drawbacks. It is difficult to almost impossible to insert the Virtualization Appliance in the data path without visibly interrupting user and/or application access to the back-end servers. Removing the Virtualization Appliance without disruption is as difficult as placing it in-line.

There are some situations, such as in an I/O intensive environment, where the latency introduced by the in-band file virtualization is deemed not acceptable. On the other hand, only in-band file virtualization offers non-disruptive reconfiguration of a namespace without shutting down all file servers that are affected by the changes during the namespace reconfiguration. Thus, if users are willing not to use the full-features provided by the in-band file virtualization, it is desirable to have a file virtualization solution that is out-of-band during normal operation and in-band only while the namespace is being reconfigured. Such a solution extends in-band file virtualization's benefit of non-disruptive namespace reconfiguration to all file servers.

When file virtualization is about to be implemented, the administrator faces the challenge of inserting the virtualization appliance without or with very limited interruption to user's access to the backend servers. By combining the knowledge of the back-end servers load, the DFS ability to redirect user access to a newly designated target, and the ability to force a user disconnect, the administrator is able to eliminate the user interruption and only in a very few cases cause an interim disruption the access of the user to the back end servers when a Virtualization Appliance is inserted in the data path between the clients machine(s) and the backend servers.

In accordance with one aspect of the invention there is provided a method for inserting a file virtualization appliance for maintaining consistency of the namespace during namespace reconfiguration in a storage network having one or more storage servers and having a distributed file system (DFS) server that exports a global namespace consisting of file objects exported by the storage servers in the storage network, and wherein clients of the storage network always consult the DFS server for the identification of a storage server that exports an unknown file object before accessing, and wherein clients of the storage network may choose to access a known file object directly from its storage server without consulting the DFS server for its accuracy. The method involves configuring a global namespace of the virtualization appliance to match a global namespace exported by the distributed filesystem server; and updating the distributed filesystem server to redirect client requests associated with the global namespace to the virtualization appliance.

In various alternative embodiments, the method may further involve, after updating the distributed filesystem server, ensuring that no clients are directly accessing the file servers; and thereafter sending an administrative alert to indicate that insertion of the virtualization appliance is complete. Ensuring that no clients are directly accessing the file servers may involve identifying active client sessions running on the file servers; and ensuring that the active client sessions include only active client sessions associated with the virtualization appliance. The virtualization appliance may be associated with a plurality of IP addresses, and ensuring that the active client sessions include only active client sessions associated with the virtualization appliance may involve ensuring that the active client sessions include only active client sessions associated with any or all of the plurality of IP addresses. Ensuring that no clients are directly accessing the file servers may involve sending a session close command to a file server in order to terminate an active client session unrelated to the virtualization appliance. Ensuring that no clients are directly accessing the file servers may involve monitoring activity associated with active client sessions; and sending an administrative alert presenting an administrator with an option to close the active client sessions. Ensuring that no clients are directly accessing the file servers may involve sending an alert to a client associated with an active client session requesting that the client close the active client session. The method may further involve automatically reconfiguring a switch to create a VLAN for the virtualization appliance. The distributed filesystem server may be configured to follow the Distributed File System standard. Connecting a virtualization appliance to the storage network may include connecting a first switch to a second switch, wherein the first switch is connected to at least one file server; connecting the virtualization appliance to the first switch; connecting the virtualization appliance to the second switch; and for each file server connected the first switch, disconnecting the file server from the first switch and connecting the file server to the second switch.

In accordance with another aspect of the invention there is provided a method for removing a virtualization appliance logically positioned between client devices and file servers in a storage network having a distributed filesystem server. The method involves sending a global namespace from the virtualization appliance to the distributed filesystem server; and configuring the virtualization appliance to not respond to any new client connection requests received by the virtualization appliance.

In various alternative embodiments, the method may further involve disconnecting the virtualization appliance from the storage network after a predetermined final timeout period. The method may also involve for any client request associated with an active client session received by the virtualization appliance during a predetermined time window, closing the client session. The predetermined time window may be between the end of a first timeout period and the predetermined final timeout period. The distributed filesystem server may be configured to follow the Distributed File System standard.

The foregoing and advantages of the invention will be appreciated more fully from the following further description thereof with reference to the accompanying drawings wherein:

FIG. C-1 is a schematic block diagram of a three server DFS system demonstrating file access from multiple clients;

FIG. C-2 is a schematic block diagram of a virtualized three server system;

FIG. C-3 depicts the process sequence of adding the Virtualization Appliance to the network;

FIG. C-4 depicts the process sequence of removing direct access between the client machines and the back-end servers;

FIG. C-5 depicts the process sequence of restoring direct access between the client machines and back-end servers;

FIG. C-6 is a logic flow diagram for logically inserting a virtualization appliance between client devices and file servers in a storage network, in accordance with an exemplary embodiment of the present invention; and

FIG. C-7 is a logic flow diagram for removing a virtualization appliance from a storage network, in accordance with an exemplary embodiment of the present invention.

Definitions. As used in this section and related claims, the following terms shall have the meanings indicated, unless the context otherwise requires:

File Virtualization: File virtualization is a technology that separates the full name of a file from its physical storage location. File virtualization is usually implemented as a hardware appliance that is located in the data path (in-band) between clients and the file servers. For users, a file Virtualization Appliance appears as a file server that exports the namespace of a file system. From the file servers' perspective, the file Virtualization Appliance appears as just a beefed up client machine that hosts a multitude of users.

Virtualization Appliance. A "Virtualization Appliance" is a network device that performs File Virtualization. It can be in-band or out-of-band device.

DFS. Distributed File System (a.k.a. DFS) is an out-of-band solution for presenting a single hierarchical view for a set of back-end servers. When the user data is replicated among multiple servers, DFS allows the clients to access the closest server based on a server ranking system. On the other hand, DFS does not provide any data replication, so in this case some other (non-DFS) solution should be used to ensure the consistency of the user data between the different copies of user data.

Embodiments of the present invention relate generally to a method for allowing a file server, with limited interruption, to be in the data path of a file virtualization appliance when reconfiguring the namespace exported by the file server and to be out of the data path of a file virtualization appliance to avoid incurring the latency introduced by the file virtualization appliance during normal operations.

Embodiments enable file virtualization to allow on-demand addition and removal of file servers under control by the file virtualization. As a result, out-of-band file servers can enjoy the benefit of continuous availability even during namespace reconfiguration.

Default DFS Operations

FIG. C-1 demonstrates how the standard DFS based virtualization works. Client11 to Client14 are regular clients that are on the same network with the DFS server (DFS1) and the back-end servers (Server11 to Server13). The clients and the servers connect through a standard network file system protocol CIFS and/or NFS over a TCP/IP switch based network.

The Clients are accessing the global name space presented by the DFS1 server. When a client wants to access a file, the client sends its file system request to the DFS server (DFS1) which informs the client that the file is being served by another server. Upon this notification, the client forms a special DFS request asking for the file placement of the file in question. The DFS server instructs the client what portion of the file path is served by which server and where on that server this path is placed. The client stores this information in its local cache and resubmits the original request to the specified server. As long as there is an entry in its local cache, the client would never ask the DFS to resolve another reference for an entity residing within that path. The cache expiration timeout is specified by the DFS administrator and by default is set to 15 minutes. There is no way for the DFS server to revoke a cached reference or purge it from a client's cache.

Since the client implements the majority of the DFS functionality, there are some significant differences in how the cache timeout is implemented depending on the Operating System (OS) and the OS version. Some clients keep the entry in the cache for as long as there is any activity and/or an open handle on that path; other clients are a little bit stricter and do enforce the time out for any new opens that come after the timeout expires. This makes it extremely difficult to predict when the client will switch to the new references. To avoid any inconsistencies, the administrators force a reboot on the client's machines or log-in to those machines, install and run a special utility that flushes the whole DFS cache for all of the servers this client is accessing, which in turn forces the client to consult the DFS server the next time it tries to access that/any file from the global namespace.

File Virtualization Operations

FIG. C-2 illustrates the basic operations of a small virtualized system that consists of four clients (Client21 to Client24), three back-end servers (Server21 to Server23) a Virtualization Appliance, and couple of IP switches 21 and 22. When clients 21-24 try to access a file, the Virtualization Appliance 2 resolves the file/directory path to a server, a server share, and a path and dispatches the client request to the appropriate back-end server 21, 22 or 23. Since the client 21 does not have direct access to the back-end servers 21-23, the Virtualization Appliance 2 can store the files and the corresponding directories at any place and in whatever format it wants, as long as it preserves the user data. Some of the major functions include: moving user files and directories without user access interruptions, mirror the user files, load balancing, and storage utilization, among others.

Physically Adding a Virtualization Appliance to a Storage Network.

FIG. C-3 demonstrates how the virtualization device is added to the physical network. The process includes manually bringing a virtualization device and an IP switch in a close proximity to the rest of the network and manually connecting them to the network.

First, administrator connects the second switch 32 to the current one 31 and connects the Virtualization Appliance 3 to both switches and turns them on (assuming they were not already on).

At this point, the administrator can unplug the first server 31 from the original switch 31 and connect it to the second switch 32. Since the network file system protocols go over a reliable transport protocol, there would be no interruption in the user/application activities as long as this operation completes within 2 to 5 seconds.

The same operation can be repeated with the rest of the servers. Alternatively, the administrator can do the hardware reconfiguration during scheduled server shut down and this way he doesn't have to worry how fast he can perform the hardware reconfiguration.

In case the IP switch is a managed switch with available ports for connection to the Virtualization Appliance 3, the above operations (aside from connecting the Virtualization Appliance to the switch) can be performed programmatically without any physical disconnect by simply reconfiguring the switch 31 to create two separate VLANs, one to represent switch 31 and one for switch 32.

Inserting the Virtualization Appliance in the Data Path

FIG. C-4 describes the steps by which the Virtualization Appliance 4 is inserted in the data path with no interruption or minimal interruption to users.

The operation begins with the Virtualization Appliance 4 reading the DFS configuration from (DFS4, step1) configuring its global namespace to match the one exported by the DFS server 4 (step2) and updating the DFS server 4 configuration (step3) to redirect all of its global namespace to the Virtualization Appliance 4. This would guarantee that any opens after the clients cache expires would go through the Virtualization Appliance 4 (step 4).

There are several methods a Virtualization Appliance 4 can utilize to make sure that clients do not access the back-end servers. This is performed (in step5) by going to the back-end servers 41-43 and obtaining the list of user sessions established. There should be no other sessions except the sessions originated through one of the IP addresses of the Virtualization Appliance 4.

When all clients start accessing the back-end servers 41-43 through the Virtualization Appliance 4, the Virtualization Appliance 4 can send an administrative alert (e-mail, SMS, page) to indicate that the insertion has been completed, so the administrator can physically disconnect the two switches 41 and 42 (step 7). In the case of a managed switch, the Virtualization Appliance 4 can reconfigure the switch to separate the two VLANs.

In the case where there are user machines that do not want to retire a cached entry, the Virtualization Appliance can kick the user off of a predetermined server by sending a session close command (step6) to the server on which the user was logged on. This would force the user's machine to reestablish the session which triggers a refresh on the affected cache entries.

To limit the impact of the session close several methods can be implemented. If the user has no open files on that session, the session can be killed since the client does not have any state other than the session itself, which the client's machine can restore without any visible impact. If the user has been idle for a prolonged interval of time (e.g. 2 hours), this is an indication that the user session can be forcefully closed.
If time is not a big issue, the Virtual Appliance 4 can perform a survey, monitoring the amount of open files and traffic load coming from the offending users and present the administrator with the option to trigger a session close when the user has the least amount of files and/or traffic. This way, the impact on the particular user would be minimized.

Another alternative is for the Virtualization Appliance 4 is to send an e-mail/SMG/page to the offending users, requesting them to reboot if twice the maximum specified timeout has expired.

With the administrator physically disconnecting the links between the two switches (switch41 and switch42), the virtual device insertion is completed.

Removing the Virtualization Appliance from the data path

Removing the Virtualization Appliance (FIG. C-5) is significantly easier than inserting it into the network.

The process begins with the administrator physically reconnecting the two switches (switch51 and switch52, step1). After that, the virtual device restores the initial DFS configuration (step2) and stops responding to any new connection establishments. In case some changes to the back-end file and directory placements are made, the Virtualization Appliance has to rebuild the DFS configuration based on the new changes.

After a while, all clients will log off from the Virtualization Appliance and connect directly to the back-end servers (steps3,4,5,6).

In case there are clients that do not go away after twice the original DFS timeout expires, the Virtualization Appliance can start kicking users off by applying the principles used when the appliance was inserted into the data path.

When there are no more user sessions going through, the administrator can safely power-down and disconnect the Virtualization Appliance from both switches (step7 and step8).

To restore the original topology, the administrator can move the back-end servers from switch52 to switch51 (steps10,11,12). And finally, the administrator can power down switch52 and disconnect it from switch51.

FIG. C-6 is a logic flow diagram for logically inserting a virtualization appliance between client devices and file servers in a storage network, in accordance with an exemplary embodiment of the present invention. In block 602, a global namespace of the virtualization appliance is configured to match a global namespace exported by the distributed filesystem server. In block 604, the distributed filesystem server is updated to redirect client requests associated with the global namespace to the virtualization appliance. In block 606, the virtualization appliance ensures that no clients are directly accessing the file servers and in block 608 thereafter sends an administrative alert to indicate that insertion of the virtualization appliance is complete.

FIG. C-7 is a logic flow diagram for removing a virtualization appliance from a storage network, in accordance with an exemplary embodiment of the present invention. In block 702, a global namespace is sent from the virtualization appliance to the distributed filesystem server. In block 704, the virtualization appliance is configured to not respond to any new client connection requests received by the virtualization appliance. In block 706, for any client request associated with an active client session received by the virtualization appliance during a predetermined time window, the virtualization appliance closes the client session. In block 708, the virtualization appliance is disconnected from the storage network after a predetermined final timeout period.

SECTION D – FILE DEDUPLICATION USING STORAGE TIERS

In enterprises today, employees tend to keep copies of all of the necessary documents and data that they access often. This is so that they can find the documents and data easily (central locations tend to change at least every so often). Furthermore, employees also tend to forget where certain things were found (in the central location), or never even knew where the document originated (they are sent a copy of the document via email). Finally, multiple employees may each keep a copy of the latest mp3 file, or video file, even if it is against company policy.

This can lead to duplicate copies of the same document or data residing in individually owned locations, so that the individual's themselves can easily find the document. However, this also means a lot of wasted space to store all of these copies of the document or data. And these copies are often stored on more expensive (and higher performance) tiers of storage, since the employees tend not to focus on costs, but rather on performance (they will store data on the location that they can most easily remember that gives them the best performance in retrieving the data).

Deduplication is a technique where files with identical contents are first identified and then only one copy of the identical contents, the single-instance copy, is kept in the physical storage while the storage space for the remaining identical contents is reclaimed and reused. Files whose contents have been deduped because of identical contents are hereafter referred to as deduplicated files. Thus, deduplication achieves what is called "Single-Instance Storage" where only the single-instance copy is stored in the physical storage, resulting in more efficient use of the physical storage space. File deduplication thus creates a domino effect of efficiency, reducing capital, administrative, and facility costs and is considered one of the most important and valuable technologies in storage.

US patents 6389433 and 6477544 are examples of how a file system provides the single-instance-storage.

While single-instance-storage is conceptually simple, implementing it without sacrificing read/write performance is difficult. Files are deduped without the owners being aware of it. The owners of deduplicated files therefore have the same performance expectation as other files that have no duplicated copies. Since many deduplicated files are sharing one single-instance copy of the contents, it is important to prevent the single-instance copy from being modified. Typically, a file system uses the copy-on-write technique to protect the single-instance copy. When an update is pending on a deduplicated file, the file system creates a partial or full copy of the single-instance copy, and the update is allowed to proceed only after the (partial) copied data has been created and only on the copied data. The delay to wait for the creation of a (partial) copy of the single-instance data before an update can proceed introduces significant performance degradation. In addition, the process to identify and dedupe replicated files also puts a strain on file system resources. Because of the performance degradation, deduplication or single-instance copy is deemed not acceptable for normal use. In reality, deduplication is of no (obvious) benefit to the end-user. Thus, while the feature of deduplication or single-instance storage has been available in a few file systems, it is not commonly used and many file systems do not even offer this feature due to its adverse performance impact.

File system level deduplication offers many advantages for the IT administrators. However, it generally offers no direct benefits to the users of the file system other than performance degradation for those files that have been deduped. Therefore, the success of deduplication in the market place depends on reducing performance degradation to an acceptable level.

Another aspect of the file system level deduplication is that deduplication is usually done on a per file system basis. It is more desirable if deduplication is done together on one or more file systems. For example, the more file systems that are deduped together, the more chances that files with identical contents will be found and more storage space will be reclaimed. For example, if there is only one copy of file A in a file system, file A will not be deduped. On the other hand, if there is a copy of file A in another file system, then together, file A in the two file systems can be deduped. Furthermore, since there is only one single-instance copy for all of the deduplicated files from one or more file systems, the more file systems that are deduped together, the more efficient the deduplication process becomes.

Thus, it is desirable to achieve deduplication with acceptable performance. It is even more desirable to be able to dedupe across more file systems to achieve more deduplication efficiency.

In accordance with one aspect of the invention there are provided a method and an apparatus for deduplicating files in a file storage system having a primary storage tier and a secondary storage tier. In such embodiments, file deduplication involves identifying a plurality of files stored in the primary storage tier having identical file contents; copying the plurality of files to the secondary storage tier; storing in the primary storage tier a single copy of the file contents; and storing metadata for each of the plurality of files, the metadata associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier.

In various alternative embodiments, identifying the plurality of files stored in the primary storage tier having identical file contents may involve computing, for each of the plurality of files, a hash value based on the contents of the file; and identifying the files having identical file contents based on the hash values. Storing the single copy of the file contents in the primary storage tier may involve copying the file contents to a designated mirror server; and deleting the remaining file contents from each of the plurality of files in the primary storage tier. Upon a read access to one of the plurality of files, the read access may be directed to the single copy of the file contents maintained in the primary storage tier. Upon a write access to one of the plurality of files, the association between the file copy in the secondary storage tier and the single copy of the file contents stored in the primary storage tier may be broken the file copy stored in the secondary storage tier may be modified. The modified file copy subsequently may be migrated from the secondary storage tier to the primary storage tier based on a migration policy.

In other embodiments, deduplicating a selected file in the primary storage tier may involve determining whether the file contents of the selected file match the file contents of a previously deduplicated file having a single copy of file contents stored in the primary storage tier; when the file contents of the selected file match the file contents of a previously deduplicated file, deduplicating the selected file; otherwise determining whether the file contents of the selected file match the file contents of a non-duplicate file in the first storage tier; and when the file contents of the selected file match the file contents of a non-duplicate file, deduplicating both the selected file and the non-duplicate file. Determining whether the file contents of the selected file match the file contents of a previously deduplicated file may involve comparing a hash value associated with the selected file to a distinct hash value associated with each single copy of file contents stored in the primary storage tier. Deduplicating the selected file may involve copying the selected file to the secondary storage tier; deleting the file contents from the selected file; and storing metadata for the selected file, the metadata associating the file copy in the secondary storage tier with the single copy of the file contents for the previously deduplicated file stored in the primary storage tier. Deduplicating both the selected file and the non-duplicate file may involve copying the selected file and the non-duplicate file to the secondary storage tier; storing in the primary storage tier a single copy of the file contents; and storing metadata for each of the first and second selected files, the metadata associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier. Storing the single copy of the file contents for deduplicating both the selected file and the non-duplicate file may involve copying the file contents to the designated mirror server; and deleting the remaining file contents from the selected file and the non-duplicate file. Determining whether the file contents of the selected file match the file contents of a non-duplicate file in the primary storage tier may involve maintaining a list of non-duplicate files in the primary storage tier, the list including a distinct hash value for each non-duplicate file; and comparing a hash value associated with the selected file to the hash values associated with the non-duplicate files in the list, and when the file contents of the selected file do not match the file contents of any non-duplicate file, may involve adding the selected file to the list of non-duplicate files (e.g., by storing a pathname and a hash value associated with the selected file). Deduplicating both the selected file and the non-duplicate file may further involve removing the non-duplicate file from the list of non-duplicate files.

Deduplication may be implemented in a file switch or other device that manages file storage.

The foregoing features of the invention will be more readily understood by reference to the following detailed description, taken with reference to the accompanying drawings, in which:

FIG. D-1 is a logic flow diagram for file deduplication using storage tiers in accordance with an exemplary embodiment of the present invention;

FIG. D-2 is a logic flow diagram deduplicating a selected file in accordance with an exemplary embodiment of the present invention.

This section relates generally to a method for performing deduplication on a global namespace using file virtualization when the global namespace is constructed from one or more storage servers, and to enable deduplication as a storage placement policy in a tiered storage environment.

A traditional file system manages the storage space by providing a hierarchical namespace. The hierarchical namespace starts from the root directory, which contains files and subdirectories. Each directory may also contain files and subdirectories identifying other files or subdirectories. Data is stored in files. Every file and directory is identified by a name. The full name of a file or directory is constructed by concatenating the name of the root directory and the names of each subdirectory that finally leads to the subdirectory containing the identified file or directory, together with the name of the file or the directory.

The full name of a file thus carries with it two pieces of information: (1) the identification of the file and (2) the physical storage location where the file is stored. If the physical storage location of a file is changed (for example, moved from one partition mounted on a system to another), the identification of the file changes as well.

For ease of management, as well as for a variety of other reasons, the administrator would like to control the physical storage location of a file. For example, important files might be stored on expensive, high-performance file servers, while less important files could be stored on less expensive and less capable file servers.

Unfortunately, moving files from one server to another usually changes the full name of the files and thus, their identification, as well. This is usually a very disruptive process, since after the move users may not be able to remember the new location of their files. Thus, it is desirable to separate the physical storage location of a file from its identification. With this separation, IT and system administrators will be able to control the physical storage location of a file while preserving what the user perceives as the location of the file (and thus its identity).

Deduplication is of no obvious benefit to the end users of a file system. Instead of using deduplication as a management policy to reduce storage space and subsequently cause inconvenience to the end users of the deduplicated files, this invention uses deduplication as a storage placement policy to intelligently managed the storage assets of an enterprise, with relatively little inconvenience to the end users.

In embodiments of the present invention, a set of file servers is designated as tier 1 where data stored in these file servers is considered more important to the enterprise. Another (typically non-overlapping) set of file servers is designated as tier 2 storage where data stored in these file servers is considered less important to the business. By using these two storage tiers to identify data important to the business, the system administrators can spend more time and resources to provide faster access and more frequent backup on the data stored on the tier 1 file servers.

Deduplication typically is treated as one of the storage placement policies that decides where data should be stored, e.g., on a tier 1 or tier 2 file server.

In embodiments of the present invention, duplicated data is automatically moved from tier 1 to tier 2. The total storage space used by the deduplicated data on tier 1 and tier 2 remains the same (or perhaps even increases slightly). However, there is more storage space available on tier 1 file servers as a result of deduplication, since all the duplicated data is now stored on tier 2.

There may be performance differences between tier 1 and tier 2 file servers. However, these differences tend to be small since the relatively inexpensive file servers are still very capable. To maintain the same level of performance when accessing the deduplicated files, as each set of duplicated files is moved from the tier 1 file servers, a single instance copy of the file is left behind as a mirror copy. One of the tier 1 file servers is designated as a mirror server where all of the mirror copies are stored. Read access to a deduplicated file is redirected to the deduplicated file's mirror copy. When the first write to a deduplicated file is received, the association from the deduplicated file stored in a tier 2 server to its mirror copy that is stored in a tier 1 server is discarded. Accesses to the "modified" duplicated file will then resume normally from the tier 2 file server. At a certain time, the "modified" deduplicated file is then migrated back to tier 1 storage.

Extending file virtualization to support deduplication is relatively straight forward. First, a set of tier-1 servers is identified as a target for deduplication, and a set of tier 2 servers is identified for receiving deduplicated data. One of the tier 1 file servers is chosen as the mirror server. The mirror server is used to store the mirror copy of each set of deduplicated files with identical contents.

A background deduplication process typically is run periodically within the file virtualization appliance to perform the deduplication. Exemplary embodiments use a sha1 digest computed from the contents of a file to identify files that have identical contents. A sha1 digest value is a 160-bit globally unique value for any given set of data (contents) of a file. Therefore, if two files are identical in contents (but not necessarily name or location), they should always have the same sha1 digest values. And conversely, if two files are different in contents, they should always have different sha1 digest values.

An exemplary deduplication process for the namespace is as follows:

1) Each file stored in the tier 1 file servers that is idle is inspected. If the file has already been deduped, it is skipped.

2) If the file does not have a sha1 digest value, it is computed and saved in the metadata for the file.

3) A check is made if there is a mirror copy stored in the mirror server. If there is, the file is deduped, and this algorithm loops around again with the next file on the tier 1 file servers.

4) The sha1 digest value and the path name of the file are then added to an internal list. If there is no existing entry in the internal list with an identical sha1 digest value, the entry is added and this algorithm loops around again with the next file on the tier 1 file servers..

5) If there is already an entry in the list with the identical sha1 digest value, the current file, as well as the other file with the same sha1 digest value listed in the internal list, will both be individually deduped and the entry in the internal list is removed. This algorithm then loops around with the next file on the tier 1 file servers.

6) The deduplicated process will continue until all the files in the tier 1 storage are processed.

It is possible that the sha1 digest value for a file marked for deduplication may have changed before it is actually deduped. This case should occur relatively infrequently. If it does occur, essentially the worst that can happen is that a file that really has no duplicate files in tier 1 gets deduplicated and migrated to tier 2. However, the deduplicated file eventually should be migrated back to the tier 1 storage tier.

An exemplary process to dedupe a single file (called from the deduplication process for the namespace) is as follows:

1) A check is made to see if there is a mirror copy with an identical sha1 digest.

2) If there is no mirror copy in the mirror server, a new mirror is made with the sha1 digest and the associated file's contents.

3) If there already is a mirror copy, the file is migrated to a tier 2 file server according to the storage placement policy. The migrated file is marked as deduplicated, and a mirror association is created between the migrated file and its mirror copy.

When a non-deduplicated file that has a sha1 digest is opened for update, its sha1 digest is immediately cleared.

When a deduplicated file is opened for update, its sha1 digest is immediately cleared. The mirror association between the deduplicated copy and the mirror copy is immediately broken. The file is no longer a deduplicated file (its deduplicated flag is cleared), and an entry is added to a to-do list to migrate this file back to tier 1 storage in the future.

When a deduplicated file is open for read, a check is made to see if there is a mirror copy stored in the mirror server. If there is, subsequent read requests on the deduplicated file will be switched to the mirror server for processing. Otherwise, the read request is switched to the tier 2 file server containing the actual data of the deduplicated file.

FIG. D-1 is a logic flow diagram for file deduplication using storage tiers in accordance with an exemplary embodiment of the present invention. In block 202, a deduplication device (e.g., a file switch) identifies a plurality of files stored in the primary storage tier having identical file contents. In block 204, the deduplication device copies the plurality of files to the secondary storage tier. In block 206, the deduplication device stores in the primary storage tier a single copy of the file contents. In block 208, the deduplication device stores metadata for each of the plurality of files, the metadata associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier.

FIG. D-2 is a logic flow diagram deduplicating a selected file in the primary storage tier in accordance with an exemplary embodiment of the present invention. In block 302, the deduplication device, determines whether the file contents of the selected file match the file contents of a previously deduplicated file having a single copy of file contents stored in the primary storage tier. When the file contents of the selected file match the file contents of a previously deduplicated file (YES in block 304), then the deduplication device deduplicates the selected file in block 306, for example, by copying the selected file to the secondary storage tier, deleting the file contents from the selected file, and storing metadata for the selected file associating the file copy in the secondary storage tier with the single copy of the file contents for the previously deduplicated file stored in the primary storage tier. When the file contents of the selected file do not match the file contents of any previously deduplicated file (NO in block 304), then the deduplication device determines whether the file contents of the selected file match the file contents of a non-duplicate file in the first storage tier in block 308. When the file contents of the selected file match the file contents of a non-duplicate file (YES in block 310), then the deduplication device deduplicates both the selected file and the non-duplicate file, for example, by copying the selected file and the non-duplicate file to the secondary storage tier, storing in the primary storage tier a single copy of the file contents, and storing metadata for each of the first and second selected files associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier. When the file contents of the selected file do not match the file contents of any non-duplicate file (NO in block 310), then the deduplication device may add the selected file a list of non-duplicate files.

It should be noted that file deduplication as discussed herein may be implemented using a file switches of the types described above and in the provisional patent application referred to by Attorney Docket No. 3193/114. It should also be noted that embodiments of the present invention may incorporate, utilize, supplement, or be combined with various features described in one or more of the other referenced patent applications.

SECTION E – FILE DEDUPLICATION USING COPY-ON-WRITE STORAGE TIERS

Section D discloses a method of deduplication where duplicated files in one or more file servers in tier-1 storage are migrated to one or more file servers in tier-2 storage. As a result, the storage space occupied by duplicated files in tier-1 storage is reclaimed, while storage space in less expensive tier-2 storage is consumed for storing the duplicated files migrated from tier-1. Furthermore, a mirror copy from each set of duplicated files is left in the tier-1 storage for maintaining read performance. The performance degradation that exists on update operation on deduplicated file is eliminated since COW is not needed. While the deduplication method specified in the co-pending application does not actually save total storage space consumed by the duplicate files, it makes it easier for end-users to accept deduplication since they will experience, at most, a very minor inconvenience. Furthermore, the number of files in tier-1 storage is reduced by deduplication, resulting in faster backup of tier-1 file servers.

However, in some cases, the actual removal of all duplicated files is unlikely to cause any inconvenience to end-users. For example, the contents of music or image files are never changed once created and are therefore good candidates for deduplication. In another case, files that have not been accessed for a long time are also good candidates, since they are unlikely to be changed again any time soon.

Therefore, it would be desirable to provide deduplication of specified classes of files.

It would be desirable to achieve deduplication with acceptable performance. It is even more desirable to be able to dedupe across more file systems to achieve higher deduplication efficiency. Furthermore, to reduce inconvenience experienced by end-users due to the performance overhead of deduplication, deduplication itself should be able to be performed on a selected set of files, instead of on every file in one or more selected file servers. Finally, in the case where end-users are unlikely to experience inconvenience due to deduplication, deduplication should result in less utilization of storage space by eliminating the storage of identical file copies.

In accordance with one aspect of the invention there is provided a method and file virtualization appliance for deduplicating files using copy-on-write storage tiers. Deduplicating files involves associating a number of files from the primary storage tier with a copy-on-write storage tier having a designated mirror server and deduplicating the files associated with the copy-on-write storage tier, such deduplicating including storing in the designated mirror server of the copy-on-write storage tier a single copy of the file contents for each duplicate and non-duplicate file associated with the copy-on-write storage tier; deleting the file contents from each deduplicated file in the copy-on-write storage tier to leave a sparse file; and storing metadata for each of the files, the metadata associating each sparse file with the corresponding single copy of the file contents stored in the designated mirror server.

In various alternative embodiments, associating a number of files from the primary storage tier with a copy-on-write storage tier may involve maintaining the copy-on-write storage tier separately from the primary storage tier and migrating the number of files from the primary storage tier to the copy-on-write storage tier. Maintaining the copy-on-write storage tier separately from the primary storage tier may involve creating a synthetic namespace for the copy-on-write storage tier using file virtualization, the synthetic namespace associated with a number of file servers, and wherein migrating the number of files from the primary storage tier to the copy-on-write storage tier comprises migrating a selected set of files from the synthetic namespace to the copy-on-write storage tier. Associating a number of files from the primary storage tier with a copy-on-write storage tier alternatively may involve marking the number of files as being associated with the copy-on-write storage tier, wherein the copy-on-write storage tier is a virtual copy-on-write storage tier. Associating a number of files from the primary storage tier with a copy-on-write storage tier may involve maintaining a set of storage policies identifying files to be associated with the copy-on-write storage tier and associating the number of files with the copy-on-write storage tier based on the set of storage policies. Storing a single copy of the file contents for each duplicate and non-duplicate file may involve determining whether the file contents of a selected file in the copy-on-write storage tier match the file contents of a previously deduplicated file having a single copy of file contents stored in the designated mirror server and when the file contents of the first selected file do not match the file contents of any previously deduplicated file, storing the file contents of the selected file in the designated mirror server. Determining whether the file contents of a selected file in the copy-on-write storage tier match the file contents of a previously deduplicated file having a single copy of file contents stored in the designated mirror server may involve comparing a hash value associated with the selected file to a hash values associated with the single copies of file contents for the previously deduplicated files stored in the designated mirror server.

Deduplicating files may further involve purging unused mirror copies from the designated mirror server. Purging unused mirror copies from the designated mirror server may involve suspending file deduplication operations; identifying mirror copies in the designated mirror server that are no longer in use; purging the unused mirror copies from the designated mirror server; and enabling file deduplication operations. Identifying mirror copies in the designated mirror server that are no longer in use may involve identifying mirror copies in the designated mirror server that are no longer associated with existing files associated with the copy-on-write storage tier. Identifying mirror copies in the designated mirror server that are no longer associated with existing files in the copy-on-write storage tier may involve constructing a list of hash values associated with existing files in the copy-on-write storage tier; and for each mirror copy in the designated mirror server, comparing a hash value associated with the mirror copy to the hash values in the list of hash values, wherein the mirror copy is deemed to be an unused mirror copy when the hash value associated with the mirror copy is not in the list of hash values.

The method may further involve processing open requests for files associated with the copy-on-write storage tier, such processing of open requests comprising:

receiving from a client an open request for a specified file associated with the copy-on-write storage tier;

when the specified file is a non-deduplicated file:

creating a copy-on-write file handle for the specified file;

marking the copy-on-write file handle as ready; and

returning the copy-on write file handle to the client;

when the specified file is a deduplicated file having a mirror copy of the file contents stored in the designated mirror server:

opening the specified file;

creating a copy-on-write file handle for the specified file;

marking the copy-on-write file handle as not ready;

returning the copy-on write file handle to the client;

when the open request is for read:

obtaining a mirror file handle for the mirror copy from the designated mirror server;

associating the mirror file handle with the copy-on-write file handle;

opening the mirror copy;

marking the copy-on-write handle as ready, if the open mirror copy is successful; and

marking the copy-on-write handle as ready with error, if the open mirror copy is unsuccessful; and

when the open request is for update:

filling the contents of the specified file from the mirror copy of the file contents stored in the designated mirror server; and

marking the copy-on-write handle as ready.

The mirror file handle for the mirror copy may be obtained from the designated mirror server based on hash values associated with the specified file and the mirror copy.

The contents of the specified file may be filled from the copy of the file contents stored in the designated mirror server using a background task.

The method may further involve processing file requests for files associated with the copy-on-write storage tier. Such processing may involve:

receiving from the client a file request including the copy-on-write file handle;

when the copy-on-write file handle is marked as not ready:

suspending the file request until the contents of the specified file have been refilled from the mirror copy;

marking the copy-on-write file handle as ready if the contents of the specified file have been refilled successfully; and marking the copy-on-write file handle as ready with error if the contents of the specified file have been refilled unsuccessfully;

when the copy-on-write file handle is marked as ready with error, returning an error indication to the client;

when the file request is a read operation and the copy-on-write file handle is associated with a mirror file handle:

using the mirror file handle to retrieve data from the mirror copy stored in the designated mirror server; and

returning the data to the client;

when the file request is a read operation and the copy-on-write file handle is not associated with a mirror file handle:

using the copy-on-write file handle to retrieve data from the file; and

returning the data to the client;

when the file request is a write operation, using the copy-on-write file handle to write data to the file in the copy-on-write storage tier; and

otherwise sending the file request to the file virtualization appliance.

The foregoing features of the invention will be more readily understood by reference to the following detailed description, taken with reference to the accompanying drawings, in which:

FIG. E-1 is a logic flow diagram for file deduplication using copy-on-write storage tiers in accordance with an exemplary embodiment of the present invention.

Embodiments of the present invention relate generally to using a copy-on-write storage tier to reclaim storage space of all duplicated files and recreate the contents of a duplicated file from its mirror copy when an update is about to occur on the duplicated file.

A traditional file system manages the storage space by providing a hierarchical namespace. The hierarchical namespace starts from the root directory, which contains files and subdirectories. Each directory may also contain files and subdirectories identifying other files or subdirectories. Data is stored in files. Every file and directory is identified by a name. The full name of a file or directory is constructed by concatenating the name of the root directory and the names of each subdirectory that finally leads to the subdirectory containing the identified file or directory, together with the name of the file or the directory.

The full name of a file thus carries with it two pieces of information: (1) the identification of the file and (2) the physical storage location where the file is stored. If the physical storage location of a file is changed (for example, moved from one partition mounted on a system to another), the identification of the file changes as well.

For ease of management, as well as for a variety of other reasons, the administrator would like to control the physical storage location of a file. For example, important files might be stored on expensive, high-performance file servers, while less important files could be stored on less expensive and less capable file servers.

Unfortunately, moving files from one server to another usually changes the full name of the files and thus, their identification, as well. This is usually a very disruptive process, since after the move users may not be able to remember the new location of their files. Thus, it is desirable to separate the physical storage location of a file from its identification. With this separation, IT and system administrators will be able to control the physical storage location of a file while preserving what the user perceives as the location of the file (and thus its identity).

Embodiments of the present invention utilize a Copy-On-Write (COW) storage tier in which every file in any of the file servers in the storage tier is eventually deduplicated, regardless whether there is any file in the storage tier that has identical contents. This is in contrast with the typical deduplication, where only files with identical contents are deduped.

Storage policies are typically used to limit the deduplication to only a set of files selected by the storage policies that apply to a synthetic namespace comprising one or more file servers. For example, one storage policy may migrate a specified class of files (e.g., all mp3 audio and jpeg image files) to a COW storage tier. Another example is that all files that have not been referenced for a specified period of time (e.g., over six months) are migrated to a COW storage tier. Once the files are in the COW storage tier, deduplication is done on every file, regardless whether any file with duplicated contents exists.

In an exemplary embodiment, extending file virtualization to support deduplication using the COW storage tier operates generally as follows. First, a synthetic namespace is created via file virtualization, and is comprised of one or more file servers. A set of storage policies is created that selects a set of files from the synthetic namespace to be migrated to the COW storage tier.

A set of file servers are selected to be in the COW storage tier. One of the file servers in a COW storage tier will also act as a mirror server. In exemplary embodiments, a mirror server is storage that may contain the current, past, or both current and past mirror copies of the authoritative copy of files stored at the COW storage tier. In exemplary embodiments, each mirror copy in the mirror sever is associated with a hash value, e.g., identified by a 160-bit number, which is the sha1 digest computed from the contents of the mirror copy. A sha1 digest value is a globally unique value for any given set of data (contents) of a file. Therefore, if two files are identical in contents (but not necessarily name or location), they should always have the same sha1 digest values. And conversely, if two files are different in contents, they should always have different sha1 digest values.

The mirror server is a special device. While it can be written, the writing of it is only performed by the file virtualization appliance itself, and each write to a file is only done once. Users and applications only read the contents of files stored on the mirror server. Basically, the mirror server is a sort of write once, read many (WORM) device. Therefore, if the mirror server were replicated, users and applications could read from any of the mirror servers available. By replicating the mirror server, one can increase the availability (if one mirror server is unavailable, another mirror server can service the request) and performance (multiple mirror servers can respond to reads from users and applications in parallel, as well as having mirror servers that are closest to the requester service the request).

Once a file is stored in a COW storage tier, the file will eventually be deduplicated. For example, if there is no update made to any files in a COW storage tier, then after a certain duration, all files in the COW storage tier will be deduped. After a file is deduped, the file becomes a sparse file where essentially all of the file's storage space is reclaimed while all of the file's attributes, including its size, remain.

A background deduplication process typically is run periodically within the file aggregation appliance to perform the deduplication. An exemplary deduplication process for a COW storage tier is as follows:

1) Each file stored in a COW storage tier is inspected.

2) If the file is not idle, the file is skipped, and the deduplication process proceeds with the next file stored in the COW storage tier.

3) If the file has already been deduped, the file is skipped, and the deduplication process proceeds with the next file stored in the COW storage tier.

4) If the file does not have a sha1 digest value, the value is computed and saved in the metadata for the file.

5) The file is deduped.

6) If the dedupe of the single file failed with an error code, then the deduplication process logs the full name of the single file together with the error code in a log file. The deduplication process will continue with the next file stored in the COW storage tier.

7) If the dedupe of the single file returned with a success code, then this algorithm loops around again with the next file. The deduplication process will continue until all the files in the COW storage are processed.

An exemplary process to dedupe a single file (called from the deduplication process for the namespace) is as follows:

1) The sha1 digest is retrieved from the metadata of the file.

2) A check is made to see if there is a mirror copy with an identical sha1 digest in the mirror server.

3) If there is no mirror copy in the mirror server, a new mirror copy is made with the sha1 digest and the file's contents. If there is no space on the mirror server for this new mirror copy, then this dedupe of a single file fails with an error code.

4) The storage space of the original file is released, resulting in a sparse file. The deduped file is marked as deduplicated, and the dedupe process returns with a success code.

When a file in COW storage tier is opened, the open request is actually sent to the MFM that manages the COW storage tier. An exemplary process to open a file is as follows:

1) Open the COW file. If the open is not successful, an error code is returned. The open operation is complete.

2) Otherwise, the file handle from opening a file in the COW storage tier is called the COW file handle. Notice that once a COW file is deduped, it becomes a sparse file and does not contain any data.

3) If the open of the COW file is successful and if the file is not a deduped file, the COW file handle is returned and the open operation is complete.

4) If the open of the file is successful and if the file is deduped, the COW file handle is marked as not ready and this handle is returned to the user. The open operation then continues as described below:

5) If the open is for read, then the sha1 digest is retrieved from the metadata and the sha1 digest for the file is then used to obtain a mirror file handle from the mirror server. If a mirror file handle is returned, the mirror file handle is associated with the COW file handle and the COW file handle is marked as ready.
6) If the open mirror file fails, the file is marked as ready (but with error). The open operation is complete.

7) If the open is for update, a background process will be informed to fill the contents of a COW file from the file's mirror copy stored in the mirror server. The open operation is complete.

When a file request is sent to the MFM, it includes a COW file handle. Exemplary steps for handling a file identified by the COW file handle are as follows:

1) If the COW file handle is marked as not ready, the request will be suspended until the COW file handle is ready (i.e. the file to be opened is made non-sparse, and the data from the mirror copy was copied into the original file in the COW storage).

2) If the COW file handle is marked as ready (but with error), an I/O error is returned.

3) If the request is a read operation and if the mirror file handle exists, the mirror file handle is used to retrieve the data. Otherwise, the COW file handle is used to retrieve the data. The result from either the COW file or the mirror server is returned to the user.

4) If the request is a write operation, the COW file handle is used to write the data to the COW storage.

5) If the request is an I/O control call sent from the background copy process informing that the contents of a COW file has been refilled from its mirror copy, the file is marked as ready. Otherwise, the file is marked as ready (but with error). Those suspended processes waiting for the not ready flag to be cleared will be woken up and their operations resumed.

6) Otherwise, all operations are sent to the MFM and processed by MFM.

As more mirror file copies are added into the mirror server, the past mirror file copies will need to be purged from the mirror server or the mirror server will eventually run out of storage space. An exemplary process to purge past mirror copies from the mirror server is as follows:

1) If the deduplication process is running, terminate that process and try again later.

2) Set up a lock to prevent the deduplication process from running.

3) Construct a list of in-use mirrors as follows:

a) Each file stored in a COW storage tier is inspected.

b) If the file is not idle, the file is skipped, and the purge process proceeds with the next file stored in the COW storage tier.

c) If the file does not have a sha1 digest value, the file is skipped, and the purge process proceeds with the next file stored in the COW storage tier.

d) Obtain the sha1 digest value from the file and add this value to the in-use mirror list.

e) This algorithm loops around again with the next file. The purge process will continue until all the files in the COW storage are processed.

4) After the in-user mirror list is constructed, the process to locate and purge past mirror file copies from the mirror server is as follows:

a) Each mirror copy stored in a COW storage tier is inspected.

b) Obtain the sha1 digest value of the mirror.

c) If the sha1 digest value is not found in the in-user mirror list, purge the mirror from the mirror server

d) This algorithm loops around again with the next mirror. The purge process will continue until all of the mirror copies in the mirror server are processed.

5) The lock to prevent the deduplication process from running is released.

Some enterprises or locations may not have multiple storage tiers available to setup a copy-on-write storage tier, or not have enough available storage in an available tier to store the large amount of mp3 and image files that a storage policy would dictate be stored on the copy-on-write storage tier. A new storage tier is just that, a new storage tier to create and manage.

Therefore, an alternative embodiment removes the restriction that the copy-on-write storage tier is a separate and real physical storage tier. The copy-on-write storage tier may just be some part of another storage tier, such as tier-1 or tier-2 storage, thus becoming a virtual storage tier. Rather than copying files to an actual storage tier, files could be marked as a part of the virtual storage tier by virtue of a metadata flag, hereafter referred to as the COW flag. If the COW flag is false, the file is just a part of the storage tier the file resides within. If the COW flag is true, the file is not part of the storage tier the file resides within. Rather, the file is part of the virtual copy-on-write storage tier.

Some advantages of this approach are that the files need not be copied to a physical tier of storage first, before deduplication. Furthermore, the IT administrator continues to just manage a single tier (or the same number of tiers as they were managing previously).

In addition to these advantages, all of the advantages of a physically separate COW tier discussed above generally continue to hold, including achieving deduplication with acceptable performance, the ability to dedupe across more file systems to achieve higher deduplication efficiency, and reducing the inconvenience experienced by end-users due to the performance overhead of deduplication based on a storage policy of deduping a selected set of files, while still resulting in less utilization of storage space by eliminating the storage of identical file copies.

As before, every file within the virtual copy-on-write storage tier will eventually be deduped, regardless whether there is any file in the virtual storage tier that has identical contents. This is in contrast with the typical deduplication, where only files with identical contents are deduped.

As above, a set of storage policies is created that selects a set of files from the synthetic namespace to be migrated to the virtual COW storage tier. If the files already reside on the tier which co-resides with the virtual COW storage tier, then no actual migration is performed. Rather, the COW flag within the metadata indicating that the file has been migrated to the virtual COW storage tier is set. If the file resides on a different storage tier than the virtual COW storage tier, then a physical migration is performed to the COW storage tier. Again, the COW flag within the metadata indicating that the file has been migrated to the virtual COW storage tier is set.

Alternatively, there may be a single virtual COW storage tier for all physical storage tiers within the namespace. In this case, when a storage policy indicates that a file should be migrated to the virtual COW storage tier, no physical migration is ever performed. The COW flag within the metadata indicating that the file has been migrated to the virtual COW storage is set. In this way, there generally is no need to select a set of file servers to be in the COW storage tier.

There is still the need to select one of the file servers to act as a mirror server.

Once a file is stored in the virtual COW storage tier, the file will eventually be deduped. In other words, if there is no update made to any files in a virtual COW storage tier, then after a certain duration, all files in the virtual COW storage tier will be deduped. After a file is deduped, the file becomes a sparse file where all of the file's storage space is reclaimed while all of the file's attributes, including its size, remain. Since the file just resides within a regular storage tier, the storage space that is reclaimed is the valuable tier storage space the file used to occupy.

As above, a background deduplication process typically is run periodically within the MFM to perform the deduplication. An exemplary deduplication process for a virtual COW storage tier is as follows:

1) Each file stored in the storage tier (or namespace) is inspected.

2) If the file is not in the virtual COW storage tier as indicated by the COW flag in the metadata, then the file is skipped, and the deduplication process proceeds with the next file stored in the storage tier (or namespace).

3) If the file is not idle, the file is skipped, and the deduplication process proceeds with the next file stored in the storage tier (or namespace).

4) If the file has already been deduped, the file is skipped, and the deduplication process proceeds with the next file stored in the storage tier (or namespace).

5) If the file does not have a sha1 digest value, the value is computed and saved in the metadata for the file.

6) The file is deduped.

7) If the dedupe of the single file failed with an error code, then the deduplication process logs the full name of the single file together with the error code in a log file. The deduplication process will continue with the next file stored in the storage tier (or namespace).

8) If the dedupe of the single file returned with a success code, then this algorithm loops around again with the next file. The deduplication process will continue until all the files in the storage tier (or namespace) are processed.

An exemplary process to dedupe a single file (as called by the deduplication process above) is essentially unchanged from the process described above. An exemplary process to dedupe a single file is as follows:

1) The sha1 digest is retrieved from the metadata of the file.

2) A check is made to see if there is a mirror copy with an identical sha1 digest in the mirror server.

3) If there is no mirror copy in the mirror server, a new mirror copy is made with the sha1 digest and the file's contents. If there is no space on the mirror server for this new mirror copy, then this dedupe of a single file fails with an error code.

4) The storage space of the original file is released, resulting in a sparse file. The deduped file is marked as deduplicated, and the dedupe process returns with a success code.

When a file is opened, the open request is actually sent to an MFM that manages the partition of the namespace. An exemplary process to open a file is as follows:

1) Determine if this is a COW file by checking the COW flag indicating if this file is part of the virtual COW storage tier. If not, return the results of the normal open call.

2) Open the COW file. If the open is not successful, an error code is returned. The open operation is complete.

3) Otherwise, the file handle from opening a file in the virtual COW storage tier is called the COW file handle. Notice that once a COW file is deduped, it becomes a sparse file and does not contain any data. Also notice that this COW file handle is really the normal file handle for opening the file in its normal place.

4) If the open of the COW file is successful and if the file is not a deduped file, the COW file handle is returned and the open operation is complete.

5) If the open of the file is successful and if the file is deduped, the COW file handle is marked as not ready and this handle is returned to the user. The open operation then continues as described below:

6) If the open is for read, then the sha1 digest is retrieved from the metadata and the sha1 digest for the file is then used to obtain a mirror file handle from the mirror server. If a mirror file handle is returned, the mirror file handle is associated with the COW file handle and the COW file handle is marked as ready. If the open mirror file fails, the file is marked as ready (but with error). The open operation is complete.

7) If the open is for update, a background process will be informed to fill the contents of a COW file from the file's mirror copy stored in the mirror server. The open operation is complete.

When a file request is sent to the MFM, it must include a file handle. Exemplary steps for handling a file are as follows:

1) If the file is a COW file (determined by checking the COW flag indicating COW storage tier), then continue using the file handle as the COW file handle. Otherwise, handle the file request as normal.

2) If the COW file handle is marked as not ready, the request will be suspended until the COW file handle is ready (i.e. the file to be opened is made non-sparse, and the data from the mirror copy was copied into the original file in the COW storage).

3) If the COW file handle is marked as ready (but with error), an I/O error is returned.

4) If the request is a read operation and if the mirror file handle exists, the mirror file handle is used to retrieve the data. Otherwise, the COW file handle is used to retrieve the data. The result from either the COW file or the mirror server is returned to the user.

5) If the request is a write operation, the COW file handle is used to write the data to the COW storage.

6) If the request is an I/O control call sent from the background copy process informing that the contents of a COW file has been refilled from its mirror copy, the file is marked as ready. Otherwise, the file is marked as ready (but with error). Those suspended processes waiting for the not ready flag to be cleared will be woken up and their operations resumed.

7) Otherwise, all operations are sent to the MFM and processed by the MFM.

As more mirror file copies are added into the mirror server, the past mirror file copies will need to be purged from the mirror server or the mirror server will eventually run out of storage space. An exemplary process to purge past mirror copies from the mirror server is as follows:

1) If the deduplication process is running, terminate the purge past mirror process and try again later.

2) Set up a lock to prevent the deduplication process from running.

3) Construct a list of in-use mirrors as follows:

a) Each file stored in the storage tier or namespace is inspected

b) If the file is not part of the virtual COW storage tier, the file is skipped, and the purge process proceeds with the next file in the storage tier (or namespace)

c) If the file is not idle, the file is skipped, and the purge process proceeds with the next file stored in the storage tier (or namespace).

d) If the file does not have a sha1 digest value, the file is skipped, and the purge process proceeds with the next file stored in the storage tier (or namespace).

e) Obtain the sha1 digest value from the file and add this value to the in-use mirror list.

f) This algorithm loops around again with the next file. The purge process will continue until all the files in the storage tier (or namespace) are processed.

4) After the in-user mirror list is constructed, the process to locate and purge past mirror file copies from the mirror server is performed as indicated in the co-patent application File Deduplication Using Copy-On-Write Storage Tiers:

a) Each mirror copy stored in a mirror server is inspected.

b) Obtain the sha1 digest value of the mirror.

c) If the sha1 digest value is not found in the in-use mirror list, purge the mirror from the mirror server

d) This algorithm loops around again with the next mirror. The purge process will continue until all of the mirror copies in the mirror server are processed.

5) The lock to prevent the deduplication process from running is released.

It should be noted that the in-user mirror list in an actual embodiment may be implemented as a hash table, a binary tree, or using other data structures commonly used by the people skilled in the art to achieve acceptable find performance.

As described here, it is still possible that the mirror server completely fills up (even though past mirror copies are purged). Therefore, the mirror server should be as large as possible, to accommodate at least one copy of all files that can exist in the COW storage tier. Otherwise, the mirror server may run out of space, and further deduplication will not be possible.

The related application entitled Remote File Virtualization Data Mirroring, a mechanism to purge mirror copies from the mirror server (any mirror copy can be purged at any given time, since an authoritative copy exists elsewhere) discusses a process for purging past mirror copies from the mirror server. Such purging of in-use mirror copies generally cannot be used in embodiments of the present invention. This is because a file that has been deduped in the COW storage tier only exists as a sparse file (no data in the file) and as a mirror copy. Thus, the mirror copy is actually the authoritative copy of the data contents of the deduped file. An in-use mirror copy is not purged because, among other things, it is difficult to locate and restore the contents of all the COW files that have the same identical mirror copy.

FIG. E-1 is a logic flow diagram for file deduplication using copy-on-write storage tiers in accordance with an exemplary embodiment of the present invention. In block 201, the file virtualization appliance associates a number of files from the primary storage tier with a copy-on-write storage tier having a designated mirror server. In block 203, the file virtualization appliance stores in the designated mirror server a single copy of the file contents for each duplicate and non-duplicate file associated with the copy-on-write storage tier. In block 205, the file virtualization appliance deletes the file contents from each deduplicated file in the copy-on-write storage tier to leave a sparse file. In block 207, the file virtualization appliance stores metadata for each of the files, the metadata associating each sparse file with the corresponding single copy of the file contents stored in the designated mirror server. In block 209, the file virtualization appliance purges unused mirror copies from the designated mirror server from time to time. In block 211, the file virtualization appliance processes open requests for files associated with the copy-on-write storage tier including creating COW files handles for such files. In block 213, the file virtualization appliance processes file requests for files associated with the COW storage tier based on COW file handles.

It should be noted that file deduplication as discussed herein may be implemented using a file switches of the types described above and in the provisional patent application referred to by Attorney Docket No. 3193/114. It should also be noted that embodiments of the present invention may incorporate, utilize, supplement, or be combined with various features described in one or more of the other referenced patent applications.

MISCELLANEOUS

It should be noted that terms such as "client," "server," "switch," and "node" may be used herein to describe devices that may be used in certain embodiments of the present invention and should not be construed to limit the present invention to any particular device type unless the context otherwise requires. Thus, a device may include, without limitation, a bridge, router, bridge-router (brouter), switch, node, server, computer, appliance, or other type of device. Such devices typically include one or more network interfaces for communicating over a communication network and a processor (e.g., a microprocessor with memory and other peripherals and/or application-specific hardware) configured accordingly to perform device functions. Communication networks generally may include public and/or private networks; may include local-area, wide-area, metropolitan-area, storage, and/or other types of networks; and may employ communication technologies including, but in no way limited to, analog technologies, digital technologies, optical technologies, wireless technologies (e.g., Bluetooth), networking technologies, and internetworking technologies.

It should also be noted that devices may use communication protocols and messages (e.g., messages created, transmitted, received, stored, and/or processed by the device), and such messages may be conveyed by a communication network or medium. Unless the context otherwise requires, the present invention should not be construed as being limited to any particular communication message type, communication message format, or communication protocol. Thus, a communication message generally may include, without limitation, a frame, packet, datagram, user datagram, cell, or other type of communication message.

It should also be noted that logic flows may be described herein to demonstrate various aspects of the invention, and should not be construed to limit the present invention to any particular logic flow or logic implementation. The described logic may be partitioned into different logic blocks (e.g., programs, modules, functions, or subroutines) without changing the overall results or otherwise departing from the true scope of the invention. Often times, logic elements may be added, modified, omitted, performed in a different order, or implemented using different logic constructs (e.g., logic gates, looping primitives, conditional logic, and other logic constructs) without changing the overall results or otherwise departing from the true scope of the invention.

The present invention may be embodied in many different forms, including, but in no way limited to, computer program logic for use with a processor (e.g., a microprocessor, microcontroller, digital signal processor, or general purpose computer), programmable logic for use with a programmable logic device (e.g., a Field Programmable Gate Array (FPGA) or other PLD), discrete components, integrated circuitry (e.g., an Application Specific Integrated Circuit (ASIC)), or any other means including any combination thereof. In a typical embodiment of the present invention, predominantly all of the described logic is implemented as a set of computer program instructions that is converted into a computer executable form, stored as such in a computer readable medium, and executed by a microprocessor under the control of an operating system.

Computer program logic implementing all or part of the functionality previously described herein may be embodied in various forms, including, but in no way limited to, a source code form, a computer executable form, and various intermediate forms (e.g., forms generated by an assembler, compiler, linker, or locator). Source code may include a series of computer program instructions implemented in any of various programming languages (e.g., an object code, an assembly language, or a high-level language such as Fortran, C, C++, JAVA, or HTML) for use with various operating systems or operating environments. The source code may define and use various data structures and communication messages. The source code may be in a computer executable form (e.g., via an interpreter), or the source code may be converted (e.g., via a translator, assembler, or compiler) into a computer executable form.

The computer program may be fixed in any form (e.g., source code form, computer executable form, or an intermediate form) either permanently or transitorily in a tangible storage medium, such as a semiconductor memory device (e.g., a RAM, ROM, PROM, EEPROM, or Flash-Programmable RAM), a magnetic memory device (e.g., a diskette or fixed disk), an optical memory device (e.g., a CD-ROM), a PC card (e.g., PCMCIA card), or other memory device. The computer program may be fixed in any form in a signal that is transmittable to a computer using any of various communication technologies, including, but in no way limited to, analog technologies, digital technologies, optical technologies, wireless technologies (e.g., Bluetooth), networking technologies, and internetworking technologies. The computer program may be distributed in any form as a removable storage medium with accompanying printed or electronic documentation (e.g., shrink wrapped software), preloaded with a computer system (e.g., on system ROM or fixed disk), or distributed from a server or electronic bulletin board over the communication system (e.g., the Internet or World Wide Web).

Hardware logic (including programmable logic for use with a programmable logic device) implementing all or part of the functionality previously described herein may be designed using traditional manual methods, or may be designed, captured, simulated, or documented electronically using various tools, such as Computer Aided Design (CAD), a hardware description language (e.g., VHDL or AHDL), or a PLD programming language (e.g., PALASM, ABEL, or CUPL).

Programmable logic may be fixed either permanently or transitorily in a tangible storage medium, such as a semiconductor memory device (e.g., a RAM, ROM, PROM, EEPROM, or Flash-Programmable RAM), a magnetic memory device (e.g., a diskette or fixed disk), an optical memory device (e.g., a CD-ROM), or other memory device. The programmable logic may be fixed in a signal that is transmittable to a computer using any of various communication technologies, including, but in no way limited to, analog technologies, digital technologies, optical technologies, wireless technologies (e.g., Bluetooth), networking technologies, and internetworking technologies. The programmable logic may be distributed as a removable storage medium with accompanying printed or electronic documentation (e.g., shrink wrapped software), preloaded with a computer system (e.g., on system ROM or fixed disk), or distributed from a server or electronic bulletin board over the communication system (e.g., the Internet or World Wide Web).

The present invention may be embodied in other specific forms without departing from the true scope of the invention. Any references to the "invention" are intended to refer to exemplary embodiments of the invention and should not be construed to refer to all embodiments of the invention unless the context otherwise requires. The described embodiments are to be considered in all respects only as illustrative and not restrictive.

What is claimed is:

1. In an aggregated filesystem having a cluster of file storage nodes and a distributed filesystem server (DFS) node, the file storage nodes collectively maintaining a shared storage including a plurality of non-overlapping portions, each file storage node owning at least one of the non-overlapping portions and including for each non-overlapping portion not owned by the file storage node a file virtualization link identifying another file storage node for the non-overlapping portion, the DFS node mapping each non-overlapping portion to a file storage node, a method for load sharing by the file storage nodes, the method comprising:

generating client requests by a number of client nodes, each client request identifying a non-overlapping portion and directed to a specific file storage node based on an access to the DFS server or information in a client cache; and

for each client request received by a file storage node, servicing the client request by the receiving file storage node if the receiving file storage node owns the identified non-overlapping portion and otherwise forwarding the client request by the receiving file storage node to another file storage node identified using the file virtualization links.

2. A method according to claim 1, further comprising:

migrating a specified non-overlapping portion from a source file storage node to a destination file server node.

3. A method according to claim 2, wherein the specified non-overlapping portion is migrated due to reconfiguration of the cluster.

4. A method according to claim 2, wherein the specified non-overlapping portion is migrated based on loading of the source file storage node.

5. A method according to claim 2, wherein migrating a specified non-overlapping portion from a source file storage node to a destination file server node comprises:

establishing a file virtualization link on the destination file server node, the file virtualization link identifying the file storage node that owns the non-overlapping portion;

updating the cluster resource to map the non-overlapping portion to the destination file storage node;

building metadata for the non-overlapping portion on the destination file storage node using sparse files such that all file and directory attributes of the non-overlapping portion are replicated on the destination file storage node without any data, and during such building, forwarding client requests received for the non-overlapping portion by the destination file storage node to the file storage node that owns the non-overlapping portion based on the file virtualization link;

after building the metadata for the non-overlapping portion on the destination file storage, copying data for the non-overlapping portion from the source file storage node to the destination file storage node, and during such copying, servicing metadata requests received for the non-overlapping portion by the destination file storage node using the metadata and forwarding data requests received for the non-overlapping portion by the destination file storage node to the file storage node that owns the non-overlapping portion based on the file virtualization link; and

after completion of the copying, designating the destination file storage node as the owner of the non-overlapping portion and thereafter servicing client requests received for the non-overlapping portion by the destination file storage node.

6. A method according to claim 5, wherein the destination file storage node is an existing file storage node in the cluster.

7. A method according to claim 5, wherein the destination file storage node is a new file storage node added to the cluster.

8. In an aggregated filesystem having a plurality of file storage nodes and a distributed filesystem server (DFS) node, the file storage nodes collectively maintaining a shared storage including a plurality of non-overlapping portions, each file storage node owning at least one of the non-overlapping portions and including for each non-

overlapping portion not owned by the file storage node a file virtualization link identifying another file storage node for the non-overlapping portion, the DFS node mapping each non-overlapping portion to a file storage node, a method for load sharing by a file storage node, the method comprising:

receiving, by the file storage node, a client request identifying a non-overlapping portion;

when the file storage node owns the identified non-overlapping portion, servicing the client request by the file storage node; and

when the file storage node does not own the identified non-overlapping portion, forwarding the client request by the file storage node to another file storage node identified using the file virtualization links.

9. A method according to claim 8, further comprising:

migrating a specified non-overlapping portion from another file storage node.

10. A method according to claim 9, wherein migrating a specified non-overlapping portion from another file storage node comprises:

maintaining a file virtualization link to the specified non-overlapping portion on the other file storage node;

migrating metadata for the specified non-overlapping portion from the other file storage node;

after migrating the metadata, migrating data for the specified non-overlapping portion from the other file storage node; and

after migrating the data, breaking the file virtualization link.

11. A method according to claim 10, further comprising:

while migrating the metadata, redirecting requests for the specified non-overlapping portion to the other file storage node.

12. A method according to claim 10, further comprising:

while migrating the data, servicing metadata requests for the specified non-overlapping portion from the migrated metadata and forwarding data request for the specified non-overlapping portion to the other file storage node.

13. A method according to claim 10, further comprising:

after breaking the file virtualization link, servicing requests for the specified non-overlapping portion from the migrated metadata and data.

14. A method according to claim 9, wherein migrating the specified non-overlapping portion from the other file storage node is done for at least one of load sharing and hotspot mitigation.

15. A file storage node for use in an aggregated filesystem having a plurality of file storage nodes and a distributed filesystem (DFS) node, the file storage nodes collectively maintaining a shared storage including a plurality of non-overlapping portions, each file storage node owning at least one of the non-overlapping portions and including for each non-overlapping portion not owned by the file storage node a file virtualization link identifying another file storage node for the non-overlapping portion, the DFS node mapping each non-overlapping portion to a file storage node, the file storage node comprising:

a network interface for receiving a client request identifying a non-overlapping portion; and

a processor configured to service the client request if the file storage node owns the identified non-overlapping portion and to forward the client request to another file storage node identified using the file virtualization links if the file storage node does not own the identified non-overlapping portion.

16. A file storage node according to claim 15, wherein the processor is further configured to migrate a specified non-overlapping portion from another file storage node.

17. A file storage node according to claim 16, wherein the processor is configured to migrate a specified non-overlapping portion from another file storage node by

maintaining a file virtualization link to the specified non-overlapping portion on the other file storage node; migrating metadata for the specified non-overlapping portion from the other file storage node; after migrating the metadata, migrating data for the specified non-overlapping portion from the other file storage node; and after migrating the data, breaking the file virtualization link.

18. A file storage node according to claim 17, wherein the processor is configured to redirect requests for the specified non-overlapping portion to the other file storage node while migrating the metadata.

19. A file storage node according to claim 17, wherein the processor is configured to service metadata requests for the specified non-overlapping portion from the migrated metadata and forward data request for the specified non-overlapping portion to the other file storage node while migrating the data.

20. A file storage node according to claim 17, where in the processor is configured to service requests for the specified non-overlapping portion from the migrated metadata and data after breaking the file virtualization link.

21. In a switched file system having a file switch in communication with a plurality of file servers including at least a source server and a destination server, where the source server manages a source native volume, a method for non-disruptive migration of the native volume from the source server to the destination server, the method comprising:

converting, by the file switch, the source native volume to a native with metadata volume using a local file system managed by the file switch;

converting, by the file switch, the native with metadata volume to a mirrored native with metadata volume including the source server and the destination server, the destination server including a mirror copy of the native with metadata volume;

removing, by the file switch, the source server from the mirrored native with metadata volume; and

converting, by the file switch, the mirror copy of the native with metadata volume on the destination server to a destination native volume on the destination server.

22. A method according to claim 21, wherein converting the source native volume to the native with metadata volume comprises:

for each source directory in the source native volume, creating a corresponding local directory in the local file system including metadata associated with the source directory copied from the source native volume; and

for each source file in the source native volume, creating a corresponding local sparse file in the local file system including file attributes copied from the source native volume but excluding the file contents associated with the source file.

23. A method according to claim 22, wherein the metadata associated with the source directory copied from the source native volume comprises directory security descriptors.

24. A method according to claim 22, wherein creating a local directory for a source directory comprises:

opening the source directory in the source native volume;

placing a lock on the source directory; and

creating the local directory and its metadata.

25. A method according to claim 21, wherein converting the native with metadata volume to the mirrored native with metadata volume comprises:

for each local directory, creating a corresponding destination directory in the destination server and maintaining a mapping of the local directory to a source directory pathname for the corresponding source directory in the source server and to a destination directory pathname for the corresponding destination directory in the destination server;

for each local file, creating a corresponding destination file in the destination server including file data copied from the source native volume and maintaining a mapping of the local file to a source file pathname for the corresponding source file in the source server and to a destination file pathname for the corresponding destination file in the destination server.

26. A method according to claim 25, wherein each mapping includes an indicator of the number of servers associated with the mirrored native with metadata volume.

27. A method according to claim 25, wherein removing the source server from the mirrored native with metadata volume comprises:

disabling usage of the source destination pathnames and the source file pathnames.

28. A method according to claim 25, wherein converting the mirror copy of the native with metadata volume on the destination server to a destination native volume comprises:

replicating state information for the destination directories and the destination files from the source native volume;

disabling usage of the local directories and local files; and

advertising the destination directories and destination files as a native volume.

29. A method according to claim 28, wherein converting the mirror copy of the native with metadata volume on the destination server to a destination native volume further comprises:

deleting unneeded metadata associated with the mirror copy of the native with metadata volume from the destination server.

30. A file switch for non-disruptive file migration in a switched file system having a plurality of file servers including at least a source server and a destination server, where the source server manages a source native volume, the file switch comprising:

a network interface for communication with the file servers; and

a processor coupled to the network interface and configured to convert the source native volume to a native with metadata volume using a local file system managed by the file switch; convert the native with metadata volume to a mirrored native with metadata volume including the source server and the destination server, the destination server including a mirror copy of the native with metadata volume; remove the source server from the mirrored native with metadata volume; and convert the mirror copy of the native with metadata volume on the destination server to a destination native volume on the destination server.

31. A file switch according to claim 30, wherein the processor is configured to convert the source native volume to the native with metadata volume by creating, for each source directory in the source native volume, a corresponding local directory in the local file system including metadata associated with the source directory copied from the source native volume; and creating, for each source file in the source native volume, a corresponding local sparse file in the local file system including file attributes copied from the source native volume but excluding the file contents associated with the source file.

32. A file switch according to claim 31, wherein the metadata associated with the source directory copied from the source native volume comprises directory security descriptors.

33. A file switch according to claim 31, wherein the processor is configured to create a local directory for a source directory by opening the source directory in the source native volume; placing a lock on the source directory; and creating the local directory and its metadata.

34. A file switch according to claim 30, wherein the processor is configured to convert the native with metadata volume to the mirrored native with metadata volume by creating, for each local directory, a corresponding destination directory in the destination server and maintaining a mapping of the local directory to a source directory pathname for the corresponding source directory in the source server and to a destination directory pathname for the corresponding destination directory in the destination server; and creating, for each local file, a corresponding destination file in the destination server including file data copied from the source native volume and maintaining a mapping of the local file to a source file pathname for the corresponding source file in the source server and to a destination file pathname for the corresponding destination file in the destination server.

35. A file switch according to claim 34, wherein each mapping includes an indicator of the number of servers associated with the mirrored native with metadata volume.

36. A file switch according to claim 34, wherein the processor is configured to remove the source server from the mirrored native with metadata volume by disabling usage of the source destination pathnames and the source file pathnames.

37. A file switch according to claim 34, wherein the processor is configured to convert the mirror copy of the native with metadata volume on the destination server to a destination native volume by replicating state information for the destination directories and the destination files from the source native volume; disabling usage of the local directories and local files; and advertising the destination directories and destination files as a native volume.

38. A file switch according to claim 37, wherein the processor is configured to convert the mirror copy of the native with metadata volume on the destination server to a destination native volume further by deleting unneeded metadata associated with the mirror copy of the native with metadata volume from the destination server.

39. In a storage network having one or more storage servers and having a distributed file system (DFS) server that exports a global namespace consisting of file objects exported by the storage servers in the storage network, and wherein clients of the storage network always consult the DFS server for the identification of a storage server that exports an unknown file object before accessing, and wherein clients of the storage network may choose to access a known file object directly from its storage server without consulting the DFS server for its accuracy, a method of inserting a file virtualization appliance for maintaining consistency of the namespace during namespace reconfiguration, the method comprising:

configuring a global namespace of the virtualization appliance to match a global namespace exported by the distributed filesystem server; and updating the distributed filesystem server to redirect client requests associated with the global namespace to the virtualization appliance.

40. A method according to claim 39, further comprising:

after updating the distributed filesystem server, ensuring that no clients are directly accessing the file servers; and

thereafter sending an administrative alert to indicate that insertion of the virtualization appliance is complete.

41. A method according to claim 40, wherein ensuring that no clients are directly accessing the file servers comprises:

identifying active client sessions running on the file servers; and

ensuring that the active client sessions include only active client sessions associated with the virtualization appliance.

42. A method according to claim 41, wherein the virtualization appliance is associated with a plurality of IP addresses, and wherein ensuring that the active client sessions include only active client sessions associated with the virtualization appliance comprises ensuring that the active client sessions include only active client sessions associated with any or all of the plurality of IP addresses.

43. A method according to claim 40, wherein ensuring that no clients are directly accessing the file servers comprises:

sending a session close command to a file server in order to terminate an active client session unrelated to the virtualization appliance.

44. A method according to claim 40, wherein ensuring that no clients are directly accessing the file servers comprises:

monitoring activity associated with active client sessions; and

sending an administrative alert presenting an administrator with an option to close the active client sessions.

45. A method according to claim 40, wherein ensuring that no clients are directly accessing the file servers comprises:

sending an alert to a client associated with an active client session requesting that the client close the active client session.

46. A method according to claim 40, further comprising:

automatically reconfiguring a switch to create a VLAN for the virtualization appliance.

47. A method according to claim 39, wherein the distributed filesystem server is configured to follow the Distributed File System standard.

48. A method according to claim 39, wherein connecting a virtualization appliance to the storage network includes:

connecting a first switch to a second switch, wherein the first switch is connected to at least one file server;

connecting the virtualization appliance to the first switch;

connecting the virtualization appliance to the second switch; and

for each file server connected the first switch, disconnecting the file server from the first switch and connecting the file server to the second switch.

49. A method for removing a virtualization appliance logically positioned between client devices and file servers in a storage network having a distributed filesystem server, the method comprising:

sending a global namespace from the virtualization appliance to the distributed filesystem server; and

configuring the virtualization appliance to not respond to any new client connection requests received by the virtualization appliance.

50. A method according to claim 49, further comprising:

disconnecting the virtualization appliance from the storage network after a predetermined final timeout period.

51. A method according to claim 49, further comprising:

for any client request associated with an active client session received by the virtualization appliance during a predetermined time window, closing the client session.

52. A method according to claim 51, wherein the predetermined time window is between the end of a first timeout period and the predetermined final timeout period.

53. A method according to claim 49, wherein the distributed filesystem server is configured to follow the Distributed File System standard.

54. In a file storage system having a primary storage tier and a secondary storage tier, a method of deduplicating files from the primary storage tier, the method comprising:

identifying a plurality of files stored in the primary storage tier having identical file contents;

copying the plurality of files to the secondary storage tier;

storing in the primary storage tier a single copy of the file contents; and

storing metadata for each of the plurality of files, the metadata associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier.

55. A method according to claim 54, wherein identifying the plurality of files stored in the primary storage tier having identical file contents comprises:

computing, for each of the plurality of files, a hash value based on the contents of the file; and

identifying the files having identical file contents based on the hash values.

56. A method according to claim 54, wherein storing the single copy of the file contents in the primary storage tier comprises:

copying the file contents to a designated mirror server; and

deleting the remaining file contents from each of the plurality of files in the primary storage tier.

57. A method according to claim 54, further comprising:

upon a read access to one of the plurality of files, directing the read access to the single copy of the file contents maintained in the primary storage tier.

58. A method according to claim 54, further comprising:

upon a write access to one of the plurality of files, breaking the association between the file copy in the secondary storage tier and the single copy of the file contents stored in the primary storage tier and modifying the file copy stored in the secondary storage tier.

59. A method according to claim 58, further comprising:

subsequently migrating the modified file copy from the secondary storage tier to the primary storage tier based on a migration policy.

60. A method according to claim 54, wherein deduplicating a selected file in the primary storage tier comprises:

determining whether the file contents of the selected file match the file contents of a previously deduplicated file having a single copy of file contents stored in the primary storage tier;

when the file contents of the selected file match the file contents of a previously deduplicated file, deduplicating the selected file;

otherwise determining whether the file contents of the selected file match the file contents of a non-duplicate file in the first storage tier; and

when the file contents of the selected file match the file contents of a non-duplicate file, deduplicating both the selected file and the non-duplicate file.

61. A method according to claim 60, wherein determining whether the file contents of the selected file match the file contents of a previously deduplicated file comprises:

comparing a hash value associated with the selected file to a distinct hash value associated with each single copy of file contents stored in the primary storage tier.

62. A method according to claim 60, wherein deduplicating the selected file comprises:

copying the selected file to the secondary storage tier;

deleting the file contents from the selected file; and

storing metadata for the selected file, the metadata associating the file copy in the secondary storage tier with the single copy of the file contents for the previously deduplicated file stored in the primary storage tier.

63. A method according to claim 60, wherein deduplicating both the selected file and the non-duplicate file comprises:

copying the selected file and the non-duplicate file to the secondary storage tier;

storing in the primary storage tier a single copy of the file contents; and

storing metadata for each of the first and second selected files, the metadata associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier.

64. A method according to claim 63, wherein storing the single copy of the file contents for deduplicating both the selected file and the non-duplicate file comprises:

copying the file contents to the designated mirror server; and

deleting the remaining file contents from the selected file and the non-duplicate file.

65. A method according to claim 63, wherein determining whether the file contents of the selected file match the file contents of a non-duplicate file in the primary storage tier comprises:

maintaining a list of non-duplicate files in the primary storage tier, the list including a distinct hash value for each non-duplicate file; and

comparing a hash value associated with the selected file to the hash values associated with the non-duplicate files in the list.

66. A method according to claim 65, further comprising:

when the file contents of the selected file do not match the file contents of any non-duplicate file, adding the selected file to the list of non-duplicate files.

67. A method according to claim 66, wherein adding the selected file to the list of non-duplicate files comprises:

storing a pathname and a hash value associated with the selected file.

68. A method according to claim 65, deduplicating both the selected file and the non-duplicate file further comprises:

removing the non-duplicate file from the list of non-duplicate files.

69. Apparatus for deduplicating files in a file storage system having a primary storage tier and a secondary storage tier, the apparatus comprising:

at least one communication interface for communicating with storage devices in the primary and secondary storage tiers; and

a processor configured to identify a plurality of files stored in the primary storage tier having identical file contents; copy the plurality of files to the secondary storage tier; store in the primary storage tier a single copy of the file contents; and store metadata for each of the plurality of files, the metadata associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier.

70. Apparatus according to claim 69, wherein the processor is configured to identify the plurality of files stored in the primary storage tier having identical file contents by computing, for each of the plurality of files, a hash value based on the contents of the file; and identifying the files having identical file contents based on the hash values.

71. Apparatus according to claim 69, wherein the processor is configured to store the single copy of the file contents in the primary storage tier by copying the file contents to a designated mirror server; and deleting the remaining file contents from each of the plurality of files in the primary storage tier.

72. Apparatus according to claim 69, wherein the processor is further configured to direct a read access for one of the plurality of files to the single copy of the file contents maintained in the primary storage tier.

73. Apparatus according to claim 69, wherein the processor is further configured to break the association between a specified file copy in the secondary storage tier and the single copy of the file contents stored in the primary storage tier and modify the file copy stored in the secondary storage tier upon a write access to the specified file.

74. Apparatus according to claim 73, wherein the processor is further configured to subsequently migrate the modified file copy from the secondary storage tier to the primary storage tier based on a migration policy.

75. Apparatus according to claim 69, wherein the processor is configured to deduplicate a selected file in the primary storage tier by determining whether the file contents of the selected file match the file contents of a previously deduplicated file having a single copy of file contents stored in the primary storage tier; when the file contents of the selected file match the file contents of a previously deduplicated file, deduplicating the selected file; otherwise determining whether the file contents of the selected file match the file contents of a non-duplicate file in the first storage tier; and when the file contents of the selected file match the file contents of a non-duplicate file, deduplicating both the selected file and the non-duplicate file.

76. Apparatus according to claim 75, wherein the processor is configured to determine whether the file contents of the selected file match the file contents of a previously deduplicated file by comparing a hash value associated with the selected file to a distinct hash value associated with each single copy of file contents stored in the primary storage tier.

77. Apparatus according to claim 75, wherein the processor is configured to deduplicate the selected file by copying the selected file to the secondary storage tier; deleting the file contents from the selected file; and storing metadata for the selected file,

the metadata associating the file copy in the secondary storage tier with the single copy of the file contents for the previously deduplicated file stored in the primary storage tier.

78. Apparatus according to claim 75, wherein the processor is configured to deduplicate both the selected file and the non-duplicate file by copying the selected file and the non-duplicate file to the secondary storage tier; storing in the primary storage tier a single copy of the file contents; and storing metadata for each of the first and second selected files, the metadata associating each of the file copies in the secondary storage tier with the single copy of the file contents stored in the primary storage tier.

79. Apparatus according to claim 78, wherein the processor is configured to store the single copy of the file contents for deduplicating both the selected file and the non-duplicate file by copying the file contents to the designated mirror server; and deleting the remaining file contents from the selected file and the non-duplicate file.

80. Apparatus according to claim 78, wherein the processor is configured to determine whether the file contents of the selected file match the file contents of a non-duplicate file in the primary storage tier by maintaining a list of non-duplicate files in the primary storage tier, the list including a distinct hash value for each non-duplicate file; and comparing a hash value associated with the selected file to the hash values associated with the non-duplicate files in the list.

81. Apparatus according to claim 80, wherein the processor is further configured to add the selected file to the list of non-duplicate files when the file contents of the selected file do not match the file contents of any non-duplicate file.

82. Apparatus according to claim 81, wherein the processor is configured to add the selected file to the list of non-duplicate files by storing a pathname and a hash value associated with the selected file.

83. Apparatus according to claim 80, wherein the processor is further configured to remove the non-duplicate file from the list of non-duplicate files upon deduplicating both the selected file and the non-duplicate file.

84. Apparatus according to claim 69, wherein the apparatus is a file switch, and wherein the storage devices are file servers managed in part by the file switch.

85. A method of deduplicating files from a primary storage tier by a file virtualization appliance in a file storage system, the method comprising:

associating a number of files from the primary storage tier with a copy-on-write storage tier having a designated mirror server; and

deduplicating the files associated with the copy-on-write storage tier, such deduplicating including:

storing in the designated mirror server a single copy of the file contents for each duplicate and non-duplicate file associated with the copy-on-write storage tier;

deleting the file contents from each deduplicated file in the copy-on-write storage tier to leave a sparse file; and

storing metadata for each of the files, the metadata associating each sparse file with the corresponding single copy of the file contents stored in the designated mirror server.

86. A method according to claim 85, wherein associating a number of files from the primary storage tier with a copy-on-write storage tier comprises:

maintaining the copy-on-write storage tier separately from the primary storage tier; and

migrating the number of files from the primary storage tier to the copy-on-write storage tier.

87. A method according to claim 86, wherein maintaining the copy-on-write storage tier separately from the primary storage tier comprises creating a synthetic namespace for the copy-on-write storage tier using file virtualization, the synthetic namespace associated with a number of file servers, and wherein migrating the number of files from

the primary storage tier to the copy-on-write storage tier comprises migrating a selected set of files from the synthetic namespace to the copy-on-write storage tier.

88. A method according to claim 85, wherein associating a number of files from the primary storage tier with a copy-on-write storage tier comprises:

marking the number of files as being associated with the copy-on-write storage tier, wherein the copy-on-write storage tier is a virtual copy-on-write storage tier.

89. A method according to claim 85, wherein associating a number of files from the primary storage tier with a copy-on-write storage tier comprises:

maintaining a set of storage policies identifying files to be associated with the copy-on-write storage tier; and

associating the number of files with the copy-on-write storage tier based on the set of storage policies.

90. A method according to claim 85, wherein storing in the designated mirror server a single copy of the file contents for each duplicate and non-duplicate file associated with the copy-on-write storage tier comprises:

determining whether the file contents of a selected file in the copy-on-write storage tier match the file contents of a previously deduplicated file having a single copy of file contents stored in the designated mirror server; and

when the file contents of the first selected file do not match the file contents of any previously deduplicated file, storing the file contents of the selected file in the designated mirror server.

91. A method according to claim 90, wherein determining whether the file contents of a selected file in the copy-on-write storage tier match the file contents of a previously deduplicated file having a single copy of file contents stored in the designated mirror server comprises:

comparing a hash value associated with the selected file to hash values associated with the single copies of file contents for the previously deduplicated files stored in the designated mirror server.

92. A method according to claim 85, further comprising:

purging unused mirror copies from the designated mirror server.

93. A method according to claim 92, wherein purging unused mirror copies from the designated mirror server comprises:

suspending file deduplication operations;

identifying mirror copies in the designated mirror server that are no longer in use;

purging the unused mirror copies from the designated mirror server; and

enabling file deduplication operations.

94. A method according to claim 93, wherein identifying mirror copies in the designated mirror server that are no longer in use comprises:

identifying mirror copies in the designated mirror server that are no longer associated with existing files associated with the copy-on-write storage tier.

95. A method according to claim 94, wherein identifying mirror copies in the designated mirror server that are no longer associated with existing files in the copy-on-write storage tier comprises:

constructing a list of hash values associated with existing files in the copy-on-write storage tier; and

for each mirror copy in the designated mirror server, comparing a hash value associated with the mirror copy to the hash values in the list of hash values, wherein the mirror copy is deemed to be an unused mirror copy when the hash value associated with the mirror copy is not in the list of hash values.

96. A method according to claim 85, further comprising:

receiving from a client an open request for a specified file associated with the copy-on-write storage tier;

when the specified file is a non-deduplicated file:

creating a copy-on-write file handle for the specified file;

marking the copy-on-write file handle as ready; and returning the copy-on write file handle to the client;

when the specified file is a deduplicated file having a mirror copy of the file contents stored in the designated mirror server:

opening the specified file;

creating a copy-on-write file handle for the specified file;

marking the copy-on-write file handle as not ready;

returning the copy-on write file handle to the client;

when the open request is for read:

obtaining a mirror file handle for the mirror copy from the designated mirror server;

associating the mirror file handle with the copy-on-write file handle;

opening the mirror copy;

marking the copy-on-write handle as ready, if the open mirror copy is successful; and

marking the copy-on-write handle as ready with error, if the open mirror copy is unsuccessful; and

when the open request is for update:

filling the contents of the specified file from the mirror copy of the file contents stored in the designated mirror server; and

marking the copy-on-write handle as ready.

97. A method according to claim 96, wherein the mirror file handle for the mirror copy is obtained from the designated mirror server based on hash values associated with the specified file and the mirror copy.

98. A method according to claim 96, wherein the contents of the specified file are filled from the copy of the file contents stored in the designated mirror server by a background task.

99. A method according to claim 96, further comprising:

receiving from the client a file request including the copy-on-write file handle;

when the copy-on-write file handle is marked as not ready:

suspending the file request until the contents of the specified file have been refilled from the mirror copy;

marking the copy-on-write file handle as ready if the contents of the specified file have been refilled successfully; and

marking the copy-on-write file handle as ready with error if the contents of the specified file have been refilled unsuccessfully;

when the copy-on-write file handle is marked as ready with error, returning an error indication to the client;

when the file request is a read operation and the copy-on-write file handle is associated with a mirror file handle:

using the mirror file handle to retrieve data from the mirror copy stored in the designated mirror server; and

returning the data to the client;

when the file request is a read operation and the copy-on-write file handle is not associated with a mirror file handle:

using the copy-on-write file handle to retrieve data from the file; and

returning the data to the client;

when the file request is a write operation, using the copy-on-write file handle to write data to the file in the copy-on-write storage tier; and

otherwise sending the file request to the file virtualization appliance.

100. A file virtualization appliance for deduplicating files from a primary storage tier in a file storage system, the file virtualization appliance comprising:

a network interface for communication with the file servers; and

a processor coupled to the network interface and configured to associate a number of files from the primary storage tier with a copy-on-write storage tier having a designated mirror server and to deduplicate the files associated with the copy-on-write storage tier, such deduplicating including:

storing in the designated mirror server a single copy of the file contents for each duplicate and non-duplicate file associated with the copy-on-write storage tier;

deleting the file contents from each deduplicated file in the copy-on-write storage tier to leave a sparse file; and

storing metadata for each of the files, the metadata associating each sparse file with the corresponding single copy of the file contents stored in the designated mirror server.

101. A file virtualization appliance according to claim 100, wherein the processor is configured to associate a number of files from the primary storage tier with a copy-on-write storage tier by maintaining the copy-on-write storage tier separately from the primary storage tier and migrating the number of files from the primary storage tier to the copy-on-write storage tier.

102. A file virtualization appliance according to claim 101, wherein the processor is configured to maintain the copy-on-write storage tier separately from the primary storage tier by creating a synthetic namespace for the copy-on-write storage tier using file virtualization, the synthetic namespace associated with a number of file servers, and wherein migrating the number of files from the primary storage tier to the copy-on-write storage tier comprises migrating a selected set of files from the synthetic namespace to the copy-on-write storage tier.

103. A file virtualization appliance according to claim 100, wherein the processor is configured to associate a number of files from the primary storage tier with a copy-on-write storage tier by marking the number of files as being associated with the copy-on-write storage tier, wherein the copy-on-write storage tier is a virtual copy-on-write storage tier.

104. A file virtualization appliance according to claim 100, wherein the processor is configured to associate a number of files from the primary storage tier with a copy-on-write storage tier by maintaining a set of storage policies identifying files to be associated with the copy-on-write storage tier and associating the number of files with the copy-on-write storage tier based on the set of storage policies.

105. A file virtualization appliance according to claim 100, wherein the processor is configured to store a single copy of the file contents for each duplicate and non-duplicate file associated with the copy-on-write storage tier by determining whether the file contents of a selected file in the copy-on-write storage tier match the file contents of a previously deduplicated file having a single copy of file contents stored in the designated mirror server and when the file contents of the first selected file do not match the file contents of any previously deduplicated file, storing the file contents of the selected file in the designated mirror server.

106. A file virtualization appliance according to claim 105, wherein the processor is configured to determine whether the file contents of a selected file in the copy-on-write storage tier match the file contents of a previously deduplicated file having a single copy of file contents stored in the designated mirror server by comparing a hash value associated with the selected file to hash values associated with the single copies of file contents for the previously deduplicated files stored in the designated mirror server.

107. A file virtualization appliance according to claim 100, wherein the processor is further configured to purge unused mirror copies from the designated mirror server.

108. A file virtualization appliance according to claim 107, wherein the processor is configured to purge unused mirror copies from the designated mirror server by suspending file deduplication operations; identifying mirror copies in the designated mirror server that are no longer in use; purging the unused mirror copies from the designated mirror server; and enabling file deduplication operations.

109. A file virtualization appliance according to claim 108, wherein the processor is configured to identify mirror copies in the designated mirror server that are no longer in use by identifying mirror copies in the designated mirror server that are no longer associated with existing files associated with the copy-on-write storage tier.

110. A file virtualization appliance according to claim 109, wherein the processor is configured to identify mirror copies in the designated mirror server that are no longer

associated with existing files in the copy-on-write storage tier by constructing a list of hash values associated with existing files in the copy-on-write storage tier and for each mirror copy in the designated mirror server, comparing a hash value associated with the mirror copy to the hash values in the list of hash values, wherein the mirror copy is deemed to be an unused mirror copy when the hash value associated with the mirror copy is not in the list of hash values.

111. A method according to claim 100, wherein the processor is further configured to process open requests for files associated with the copy-on-write storage tier, such processing of open requests comprising:

receiving from a client an open request for a specified file associated with the copy-on-write storage tier;

when the specified file is a non-deduplicated file:

creating a copy-on-write file handle for the specified file;

marking the copy-on-write file handle as ready; and

returning the copy-on write file handle to the client;

when the specified file is a deduplicated file having a mirror copy of the file contents stored in the designated mirror server:

opening the specified file;

creating a copy-on-write file handle for the specified file;

marking the copy-on-write file handle as not ready;

returning the copy-on write file handle to the client;

when the open request is for read:

obtaining a mirror file handle for the mirror copy from the designated mirror server;

associating the mirror file handle with the copy-on-write file handle;

opening the mirror copy;

marking the copy-on-write handle as ready, if the open mirror copy is successful; and

marking the copy-on-write handle as ready with error, if the open mirror copy is unsuccessful; and when the open request is for update:

filling the contents of the specified file from the mirror copy of the file contents stored in the designated mirror server; and

marking the copy-on-write handle as ready.

112. A method according to claim 111, wherein the processor is configured to obtain the mirror file handle for the mirror copy from the designated mirror server based on hash values associated with the specified file and the mirror copy.

113. A method according to claim 111, wherein the processor is configured to fill the contents of the specified file from the copy of the file contents stored in the designated mirror server using a background task.

114. A method according to claim 111, wherein the processor is further configured to process file requests, such processing of file requests comprising:

receiving from the client a file request including the copy-on-write file handle;

when the copy-on-write file handle is marked as not ready:

suspending the file request until the contents of the specified file have been refilled from the mirror copy;

marking the copy-on-write file handle as ready if the contents of the specified file have been refilled successfully; and

marking the copy-on-write file handle as ready with error if the contents of the specified file have been refilled unsuccessfully;

when the copy-on-write file handle is marked as ready with error, returning an error indication to the client;

when the file request is a read operation and the copy-on-write file handle is associated with a mirror file handle:

using the mirror file handle to retrieve data from the mirror copy stored in the designated mirror server; and

returning the data to the client;

when the file request is a read operation and the copy-on-write file handle is not associated with a mirror file handle:

using the copy-on-write file handle to retrieve data from the file; and

returning the data to the client;

when the file request is a write operation, using the copy-on-write file handle to write data to the file in the copy-on-write storage tier; and

otherwise sending the file request to the file virtualization appliance.

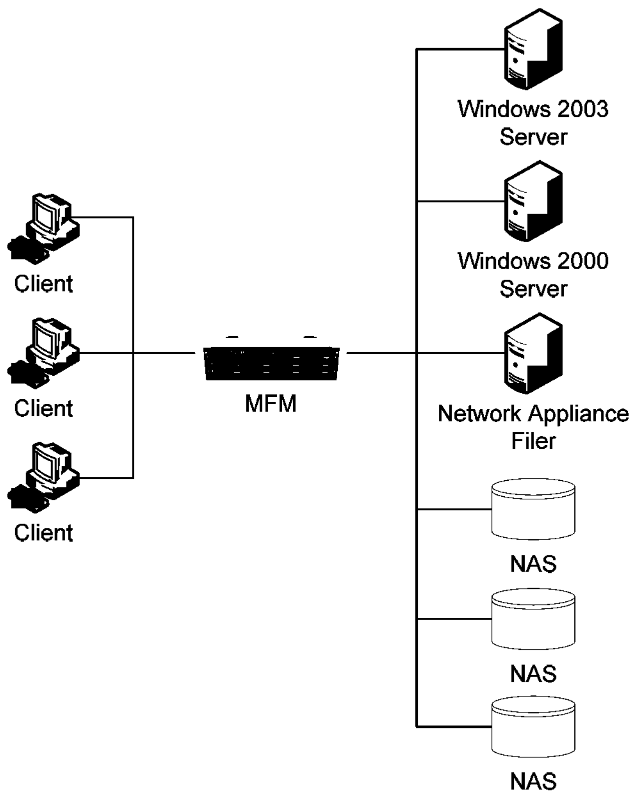


FIG. A-1 PRIOR ART

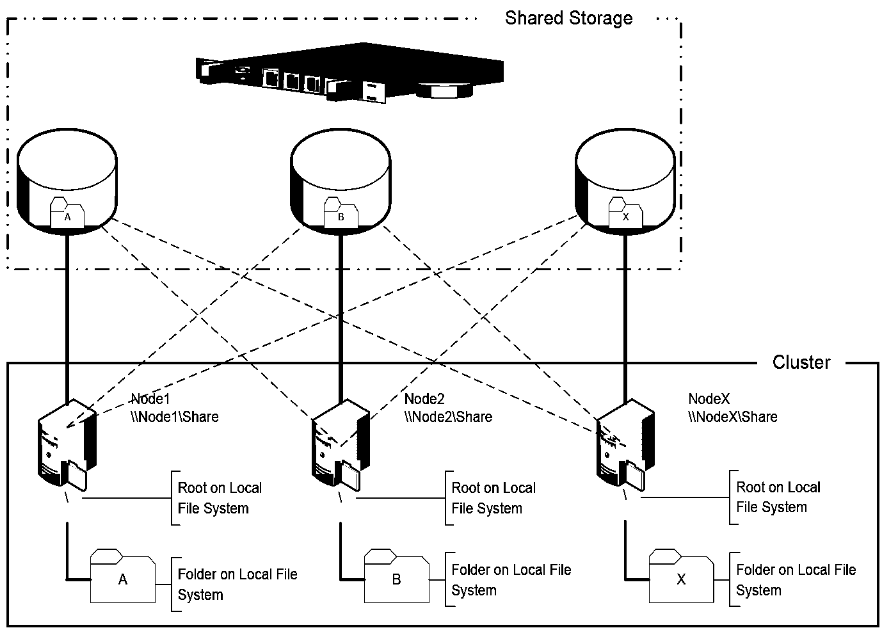


FIG. A-2 PRIOR ART

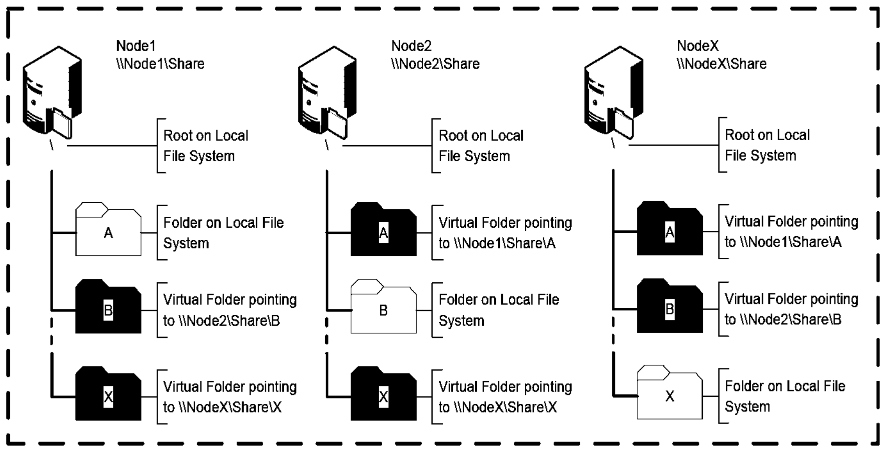

Node1
\\Node1\Share
Root on Local File System
\
A
Folder on Local File System
B
Virtual Folder pointing to \\Node2\Share\B
X
Virtual Folder pointing to \\NodeX\Share\X
Node2
\\Node2\Share
Root on Local File System
\
A
Virtual Folder pointing to \\Node1\Share\A
B
Folder on Local File System
X
Virtual Folder pointing to \\NodeX\Share\X
NodeX
\\NodeX\Share
Root on Local File System
\
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node2\Share\B
X
Folder on Local File System

FIG. A-3 PRIOR ART

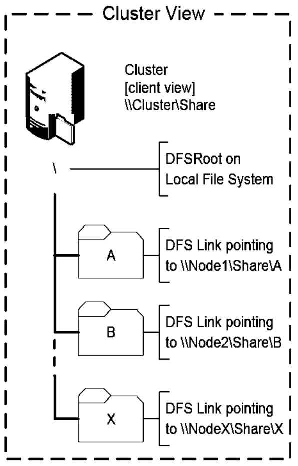


FIG. A-4 PRIOR ART

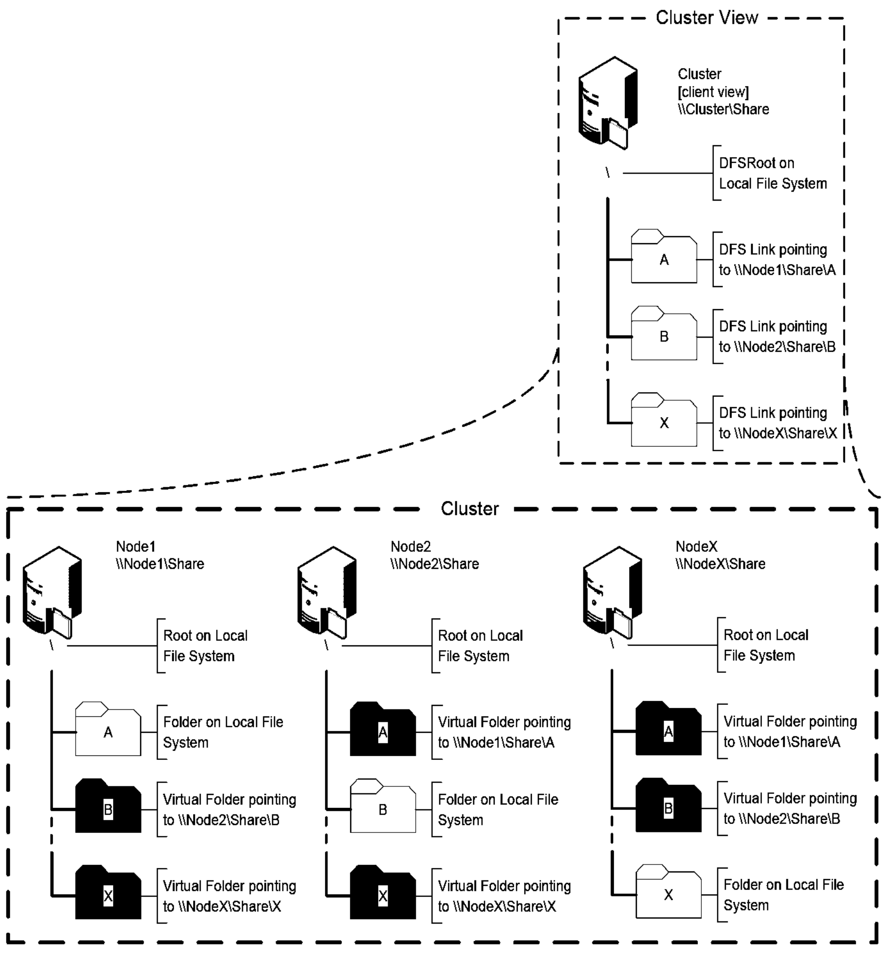

Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on Local File System
A
DFS Link pointing to \\Node1\Share\A
B
DFS Link pointing to \\Node2\Share\B
X
DFS Link pointing to \\NodeX\Share\X
Cluster
Node1
\\Node1\Share
\
Root on Local File System
A
Folder on Local File System
B
Virtual Folder pointing to \\Node2\Share\B
X
Virtual Folder pointing to \\NodeX\Share\X
Node2
\\Node2\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Folder on Local File System
X
Virtual Folder pointing to \\NodeX\Share\X
NodeX
\\NodeX\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node2\Share\B
X
Folder on Local File System

FIG. A-5

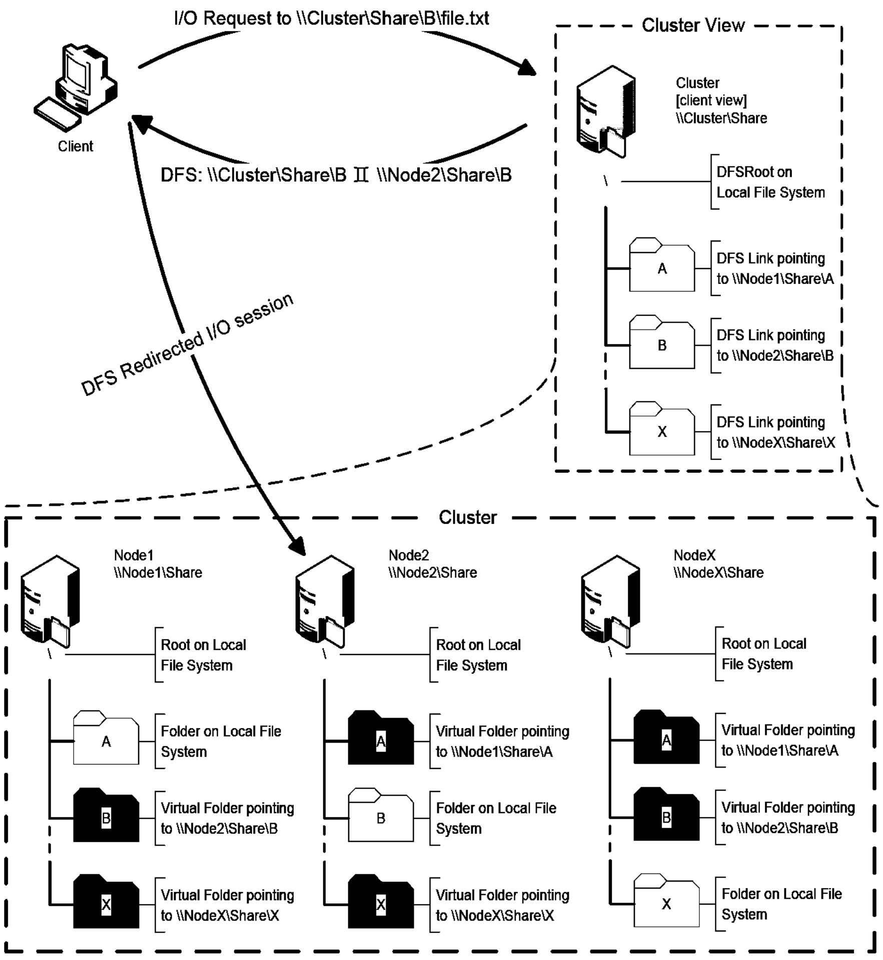

I/O Request to \\Cluster\Share\B\file.txt
Client
DFS: \\Cluster\Share\B Ⅱ \\Node2\Share\B
DFS Redirected I/O session
Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on Local File System
A
DFS Link pointing to \\Node1\Share\A
B
DFS Link pointing to \\Node2\Share\B
X
DFS Link pointing to \\NodeX\Share\X
Cluster
Node1
\\Node1\Share
\
Root on Local File System
A
Folder on Local File System
B
Virtual Folder pointing to \\Node2\Share\B
X
Virtual Folder pointing to \\NodeX\Share\X
Node2
\\Node2\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Folder on Local File System
X
Virtual Folder pointing to \\NodeX\Share\X
NodeX
\\NodeX\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node2\Share\B
X
Folder on Local File System

FIG. A-6

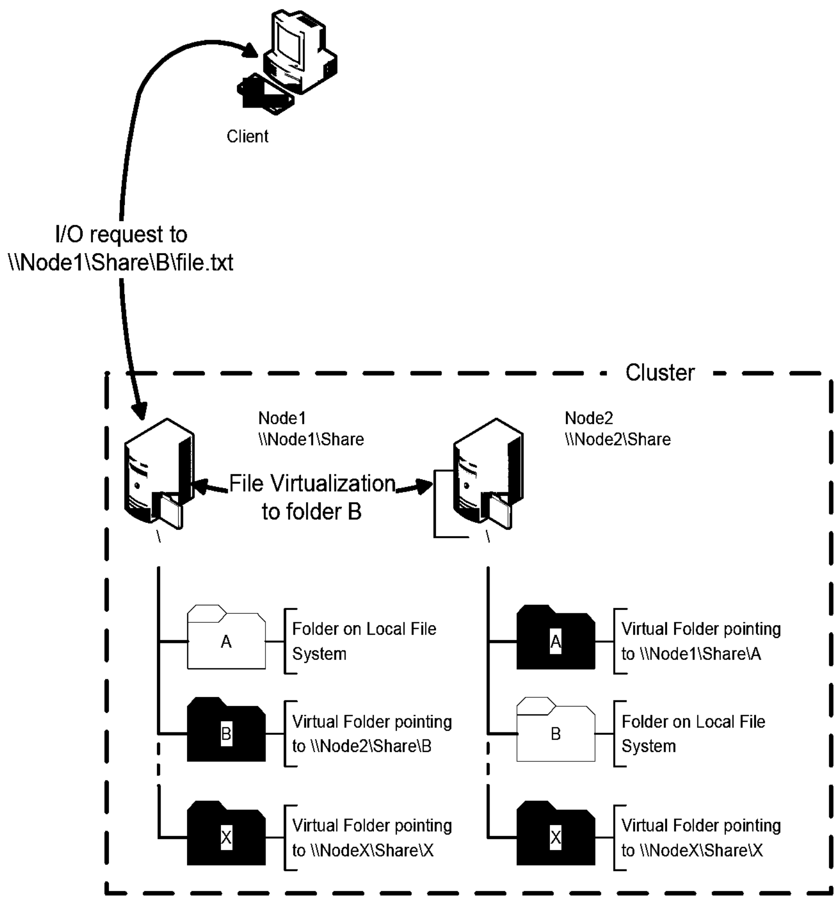

Client
I/O request to \\Node1\Share\B\file.txt
Cluster
Node1
\\Node1\Share
Node2
\\Node2\Share
File Virtualization to folder B
\
\
A
Folder on Local File System
B
Virtual Folder pointing to \\Node2\Share\B
X
Virtual Folder pointing to \\NodeX\Share\X
A
Virtual Folder pointing to \\Node1\Share\A
B
Folder on Local File System
X
Virtual Folder pointing to \\NodeX\Share\X

FIG. A-7

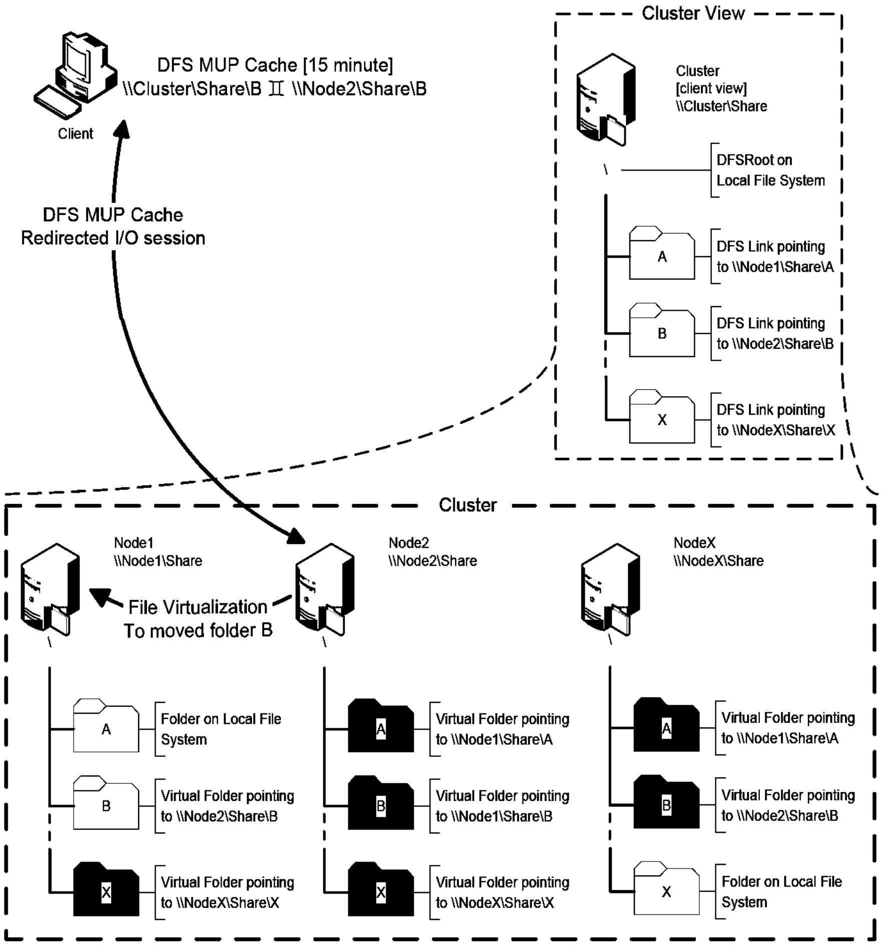

DFS MUP Cache [15 minute]
\\Cluster\Share\B ⌶ \\Node2\Share\B
Client
DFS MUP Cache
Redirected I/O session
Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on
Local File System
A
DFS Link pointing
to \\Node1\Share\A
B
DFS Link pointing
to \\Node2\Share\B
X
DFS Link pointing
to \\NodeX\Share\X
Cluster
Node1
\\Node1\Share
Node2
\\Node2\Share
NodeX
\\NodeX\Share
File Virtualization
To moved folder B
A
Folder on Local File
System
B
Virtual Folder pointing
to \\Node2\Share\B
X
Virtual Folder pointing
to \\NodeX\Share\X
A
Virtual Folder pointing
to \\Node1\Share\A
B
Virtual Folder pointing
to \\Node1\Share\B
X
Virtual Folder pointing
to \\NodeX\Share\X
A
Virtual Folder pointing
to \\Node1\Share\A
B
Virtual Folder pointing
to \\Node2\Share\B
X
Folder on Local File
System

FIG. A-8

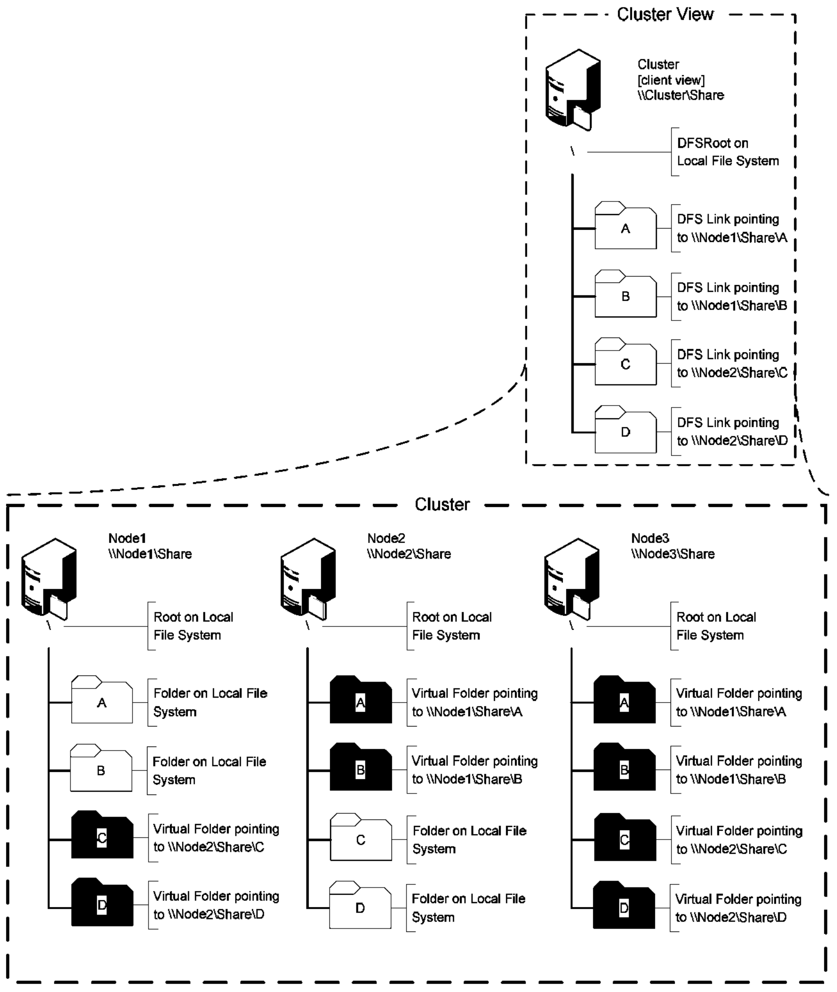

Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on Local File System
A
DFS Link pointing to \\Node1\Share\A
B
DFS Link pointing to \\Node1\Share\B
C
DFS Link pointing to \\Node2\Share\C
D
DFS Link pointing to \\Node2\Share\D
Cluster
Node1
\\Node1\Share
\
Root on Local File System
A
Folder on Local File System
B
Folder on Local File System
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node2\Share\D
Node2
\\Node2\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Folder on Local File System
D
Folder on Local File System
Node3
\\Node3\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node2\Share\D

FIG. A-9

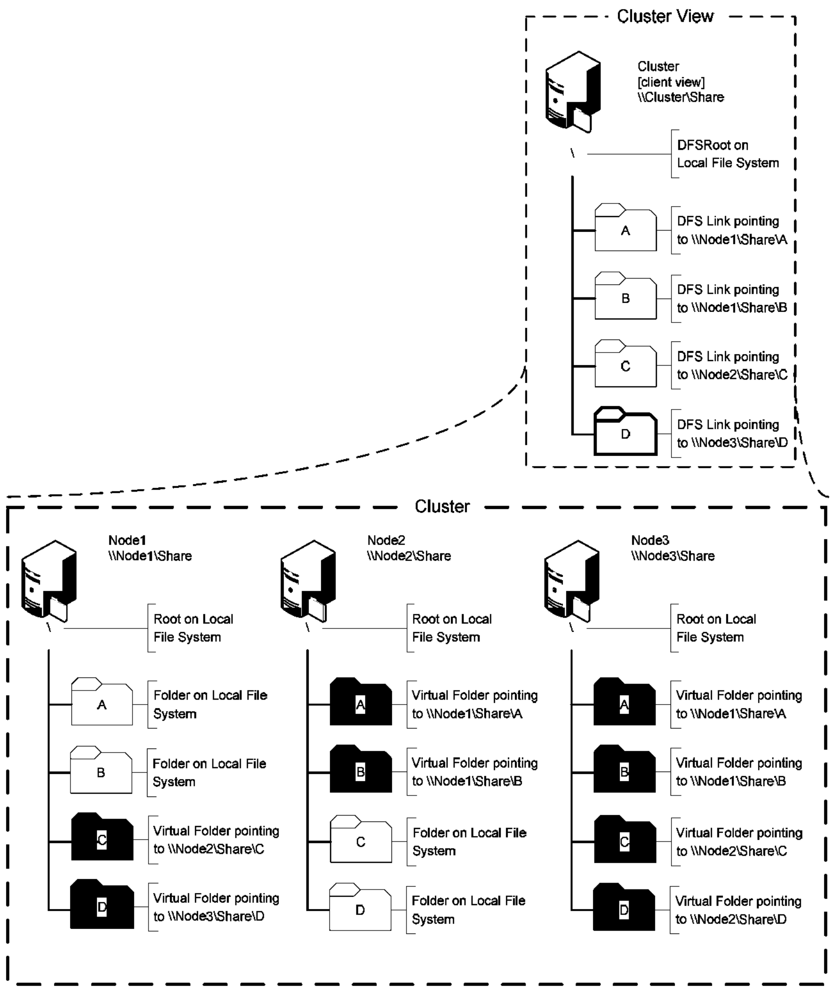

Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on Local File System
A
DFS Link pointing to \\Node1\Share\A
B
DFS Link pointing to \\Node1\Share\B
C
DFS Link pointing to \\Node2\Share\C
D
DFS Link pointing to \\Node3\Share\D
Cluster
Node1
\\Node1\Share
\
Root on Local File System
A
Folder on Local File System
B
Folder on Local File System
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node3\Share\D
Node2
\\Node2\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Folder on Local File System
D
Folder on Local File System
Node3
\\Node3\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node2\Share\D

FIG. A-10

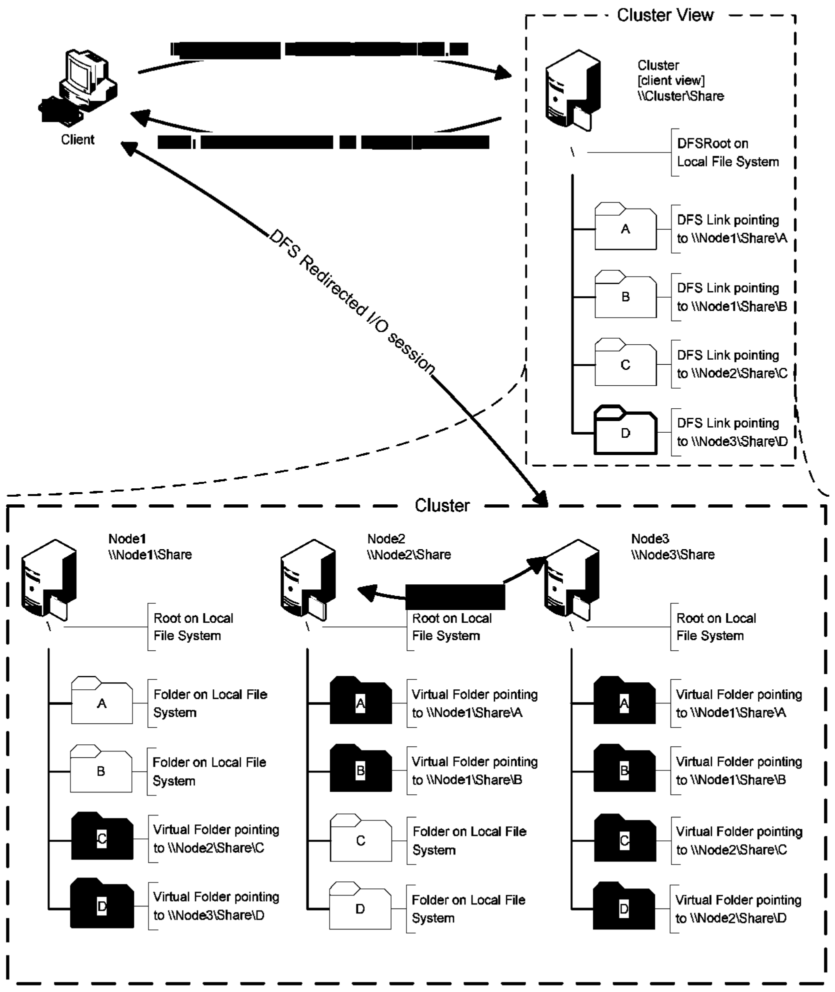

Cluster View
[redacted]
[redacted]
Client
Cluster
[client view]
\\Cluster\Share
DFSRoot on
Local File System
A
DFS Link pointing
to \\Node1\Share\A
B
DFS Link pointing
to \\Node1\Share\B
C
DFS Link pointing
to \\Node2\Share\C
D
DFS Link pointing
to \\Node3\Share\D
DFS Redirected I/O session
Cluster
Node1
\\Node1\Share
Root on Local
File System
A
Folder on Local File
System
B
Folder on Local File
System
C
Virtual Folder pointing
to \\Node2\Share\C
D
Virtual Folder pointing
to \\Node3\Share\D
Node2
\\Node2\Share
[redacted]
Root on Local
File System
A
Virtual Folder pointing
to \\Node1\Share\A
B
Virtual Folder pointing
to \\Node1\Share\B
C
Folder on Local File
System
D
Folder on Local File
System
Node3
\\Node3\Share
Root on Local
File System
A
Virtual Folder pointing
to \\Node1\Share\A
B
Virtual Folder pointing
to \\Node1\Share\B
C
Virtual Folder pointing
to \\Node2\Share\C
D
Virtual Folder pointing
to \\Node2\Share\D

FIG. A-11

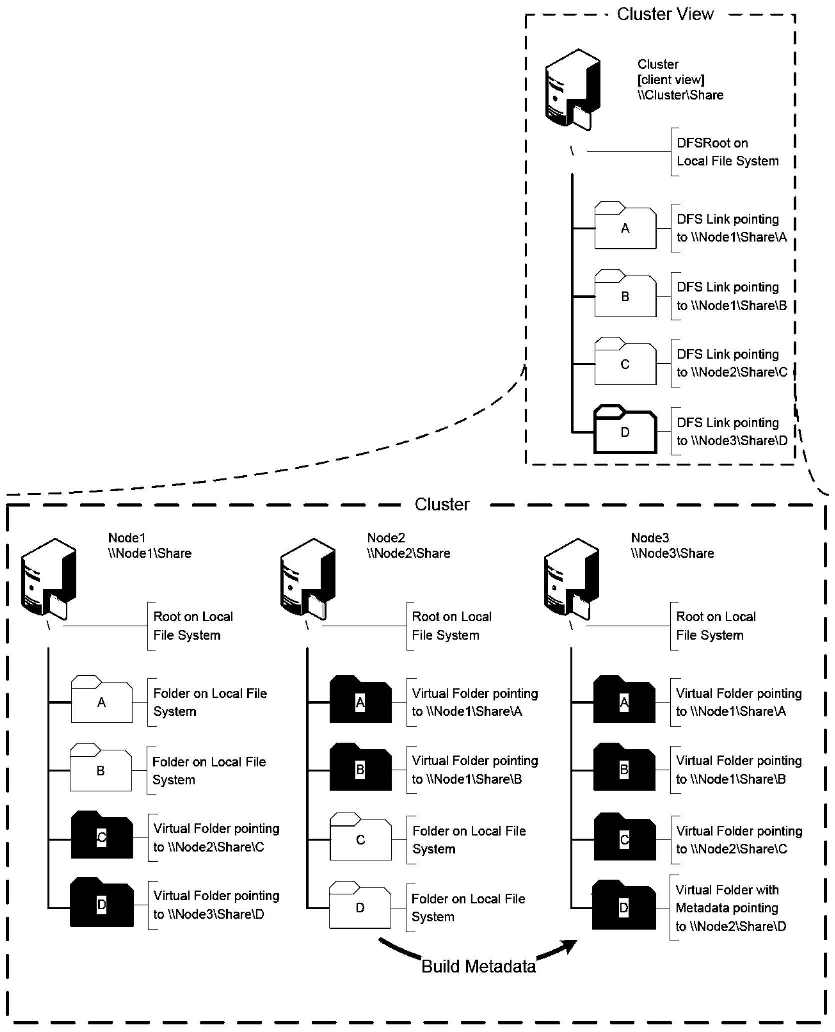

Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on Local File System
A
DFS Link pointing to \\Node1\Share\A
B
DFS Link pointing to \\Node1\Share\B
C
DFS Link pointing to \\Node2\Share\C
D
DFS Link pointing to \\Node3\Share\D
Cluster
Node1
\\Node1\Share
\
Root on Local File System
A
Folder on Local File System
B
Folder on Local File System
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node3\Share\D
Node2
\\Node2\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Folder on Local File System
D
Folder on Local File System
Build Metadata
Node3
\\Node3\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder with Metadata pointing to \\Node2\Share\D

FIG. A-12

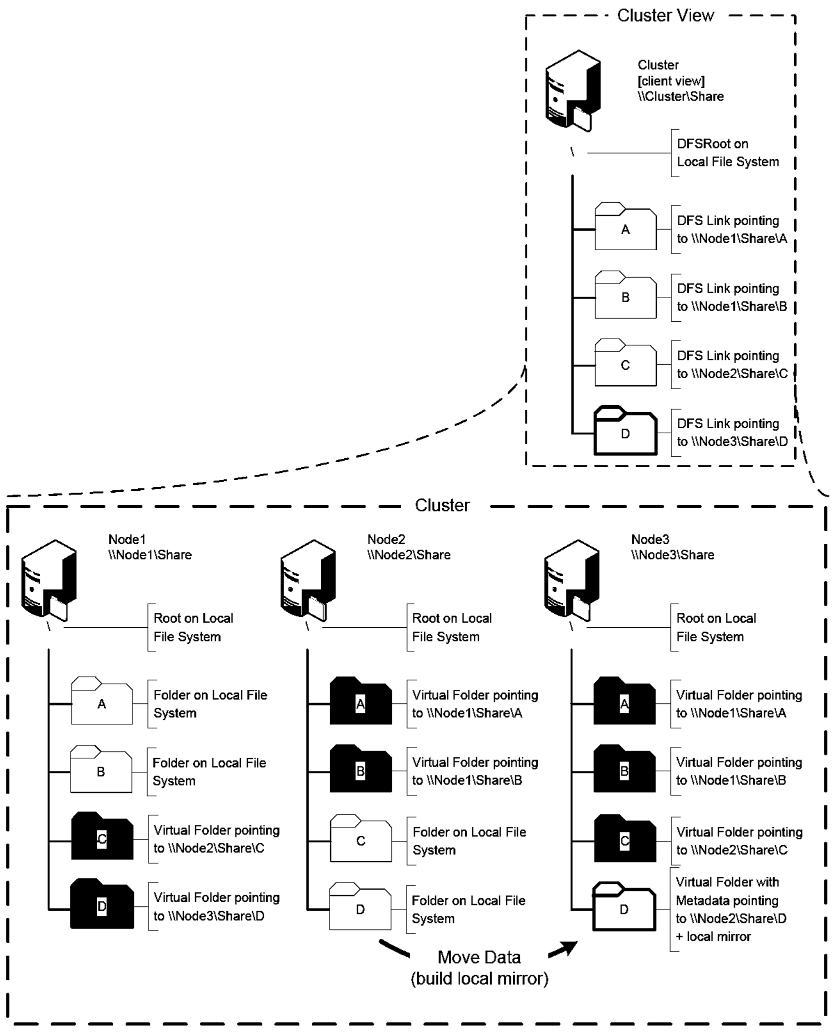

Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on Local File System
A
DFS Link pointing to \\Node1\Share\A
B
DFS Link pointing to \\Node1\Share\B
C
DFS Link pointing to \\Node2\Share\C
D
DFS Link pointing to \\Node3\Share\D
Cluster
Node1
\\Node1\Share
\
Root on Local File System
A
Folder on Local File System
B
Folder on Local File System
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node3\Share\D
Node2
\\Node2\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Folder on Local File System
D
Folder on Local File System
Node3
\\Node3\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder with Metadata pointing to \\Node2\Share\D + local mirror
Move Data (build local mirror)

FIG. A-13

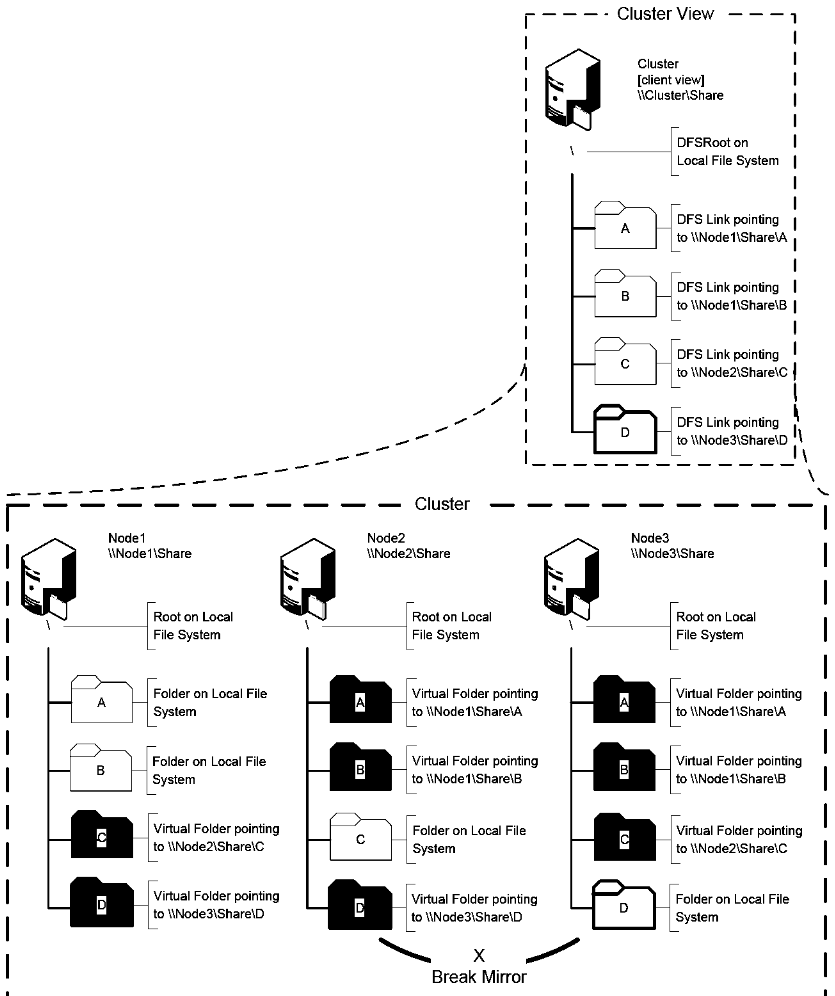

Cluster View
Cluster
[client view]
\\Cluster\Share
\
DFSRoot on Local File System
A
DFS Link pointing to \\Node1\Share\A
B
DFS Link pointing to \\Node1\Share\B
C
DFS Link pointing to \\Node2\Share\C
D
DFS Link pointing to \\Node3\Share\D
Cluster
Node1
\\Node1\Share
\
Root on Local File System
A
Folder on Local File System
B
Folder on Local File System
C
Virtual Folder pointing to \\Node2\Share\C
D
Virtual Folder pointing to \\Node3\Share\D
Node2
\\Node2\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Folder on Local File System
D
Virtual Folder pointing to \\Node3\Share\D
Node3
\\Node3\Share
\
Root on Local File System
A
Virtual Folder pointing to \\Node1\Share\A
B
Virtual Folder pointing to \\Node1\Share\B
C
Virtual Folder pointing to \\Node2\Share\C
D
Folder on Local File System
X
Break Mirror

FIG. A-14

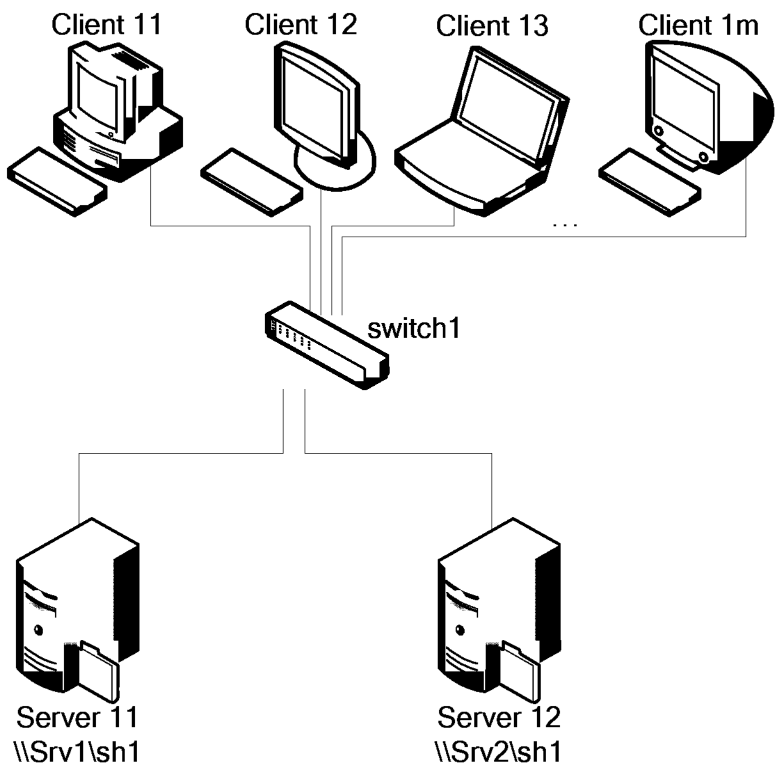


FIG. B-1 PRIOR ART

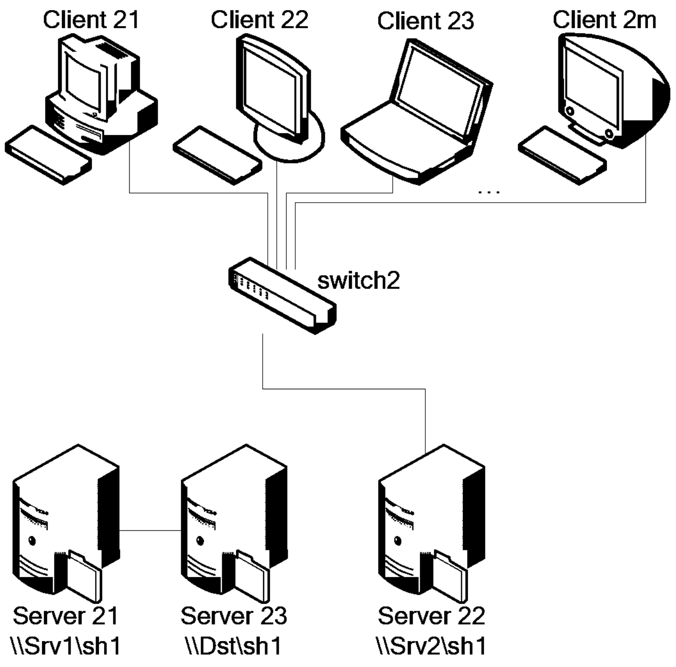


FIG. B-2 PRIOR ART

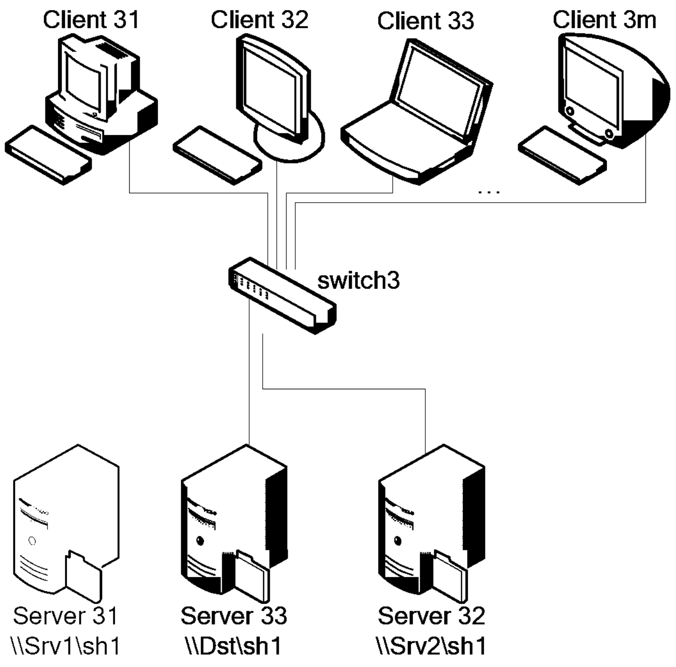


FIG. B-3 PRIOR ART

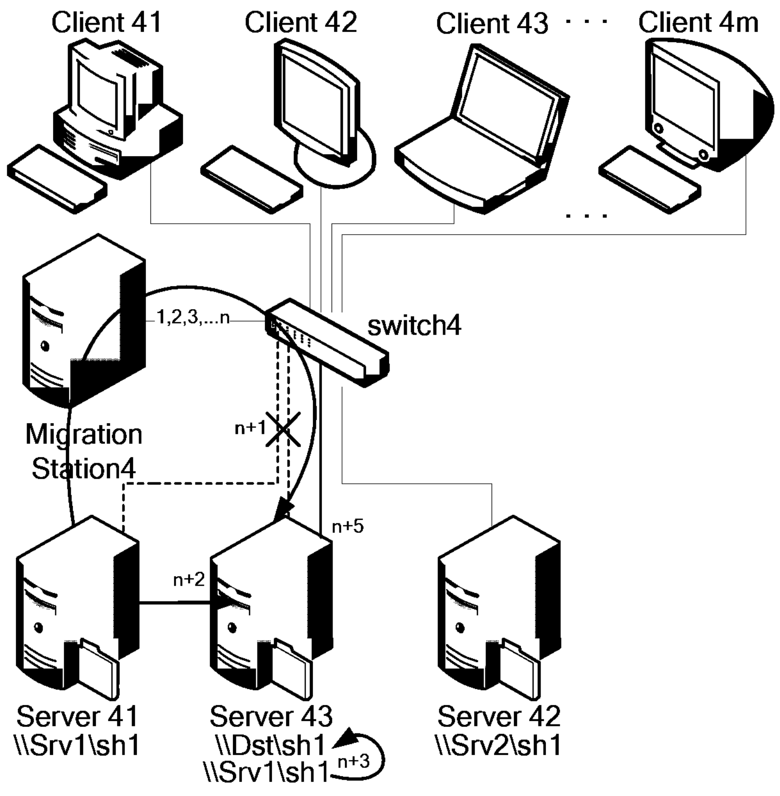


FIG. B-4 PRIOR ART

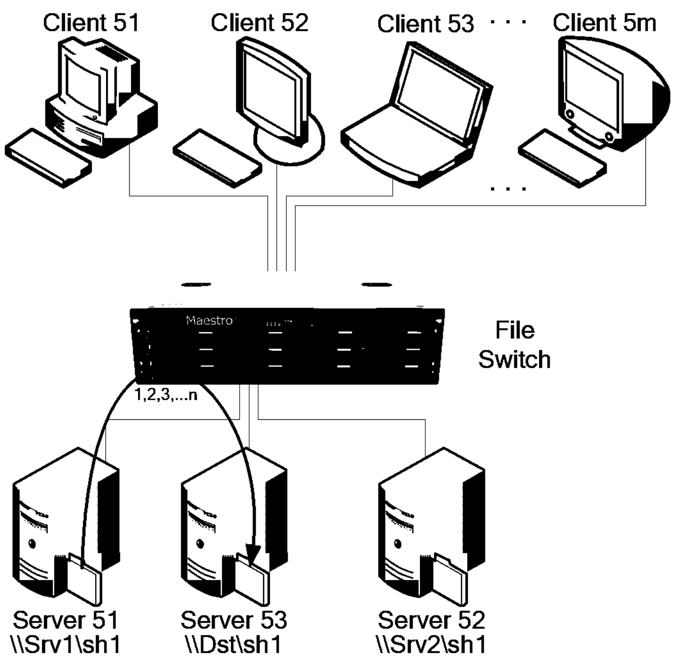


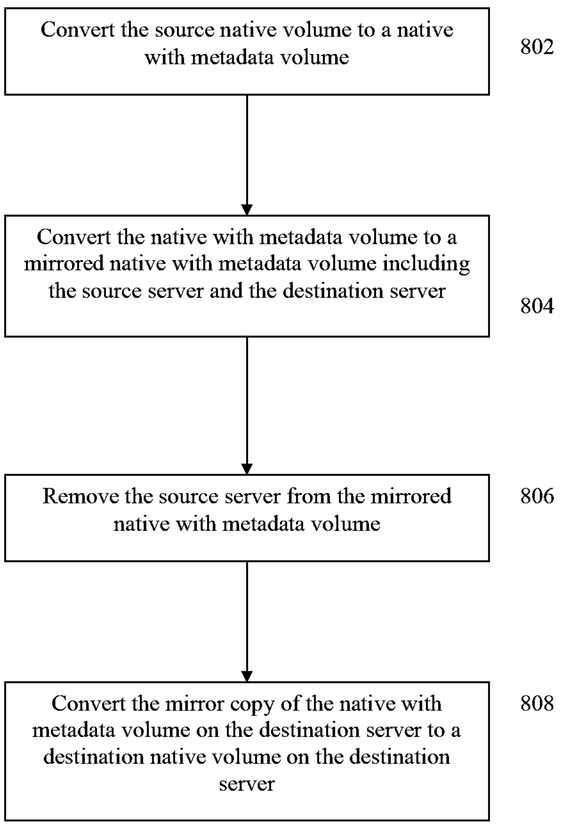
FIG. B-5

Global Namespace Path	Metadata Info	Target Path
\\MFM\Native\Srv1\Dir1	\\Srv1\sh1	\\Srv1\sh1\Dir1
\\MFM\Native\Srv1\Dir2	\\Srv1\sh1	\\Srv1\sh1\Dir2
\\MFM\Native\Srv1\Dir2\File1.TXT	\\Srv1\sh1	\\Srv1\sh1\Dir2\File1.TXT

FIG. B-6

Global Namespace Path	Metadata Info		Target Path(s)
	# Mirrors	Destination Path	
\\MFM\Native\Srv1\Dir1	1	\\Srv1\sh1	\\Srv1\sh1\Dir1
\\MFM\Native\Srv1\Dir2	1	\\Srv1\sh1	\\Srv1\sh1\Dir2
\\MFM\Native\Srv1\Dir2\File1.TXT	1	\\Srv1\sh1	\\Srv1\sh1\Dir2\File1.TXT
\\MFM\Native\Srv1\MirroredDir	2	\\Srv1\sh1 \\Dst\sh1	\\Srv1\sh1\MirroredDir \\Dst\sh1\MirroredDir
\\MFM\Native\Srv1\MirroredDir\Mirr.txt	2	\\Srv1\sh1 \\Dst\sh1	\\Srv1\sh1\MirroredDir\Mirr.txt \\Dst\sh1\MirroredDir\Mirr.txt

FIG. B-7



Convert the source native volume to a native with metadata volume
802
Convert the native with metadata volume to a mirrored native with metadata volume including the source server and the destination server
804
Remove the source server from the mirrored native with metadata volume
806
Convert the mirror copy of the native with metadata volume on the destination server to a destination native volume on the destination server
808

FIG. B-8

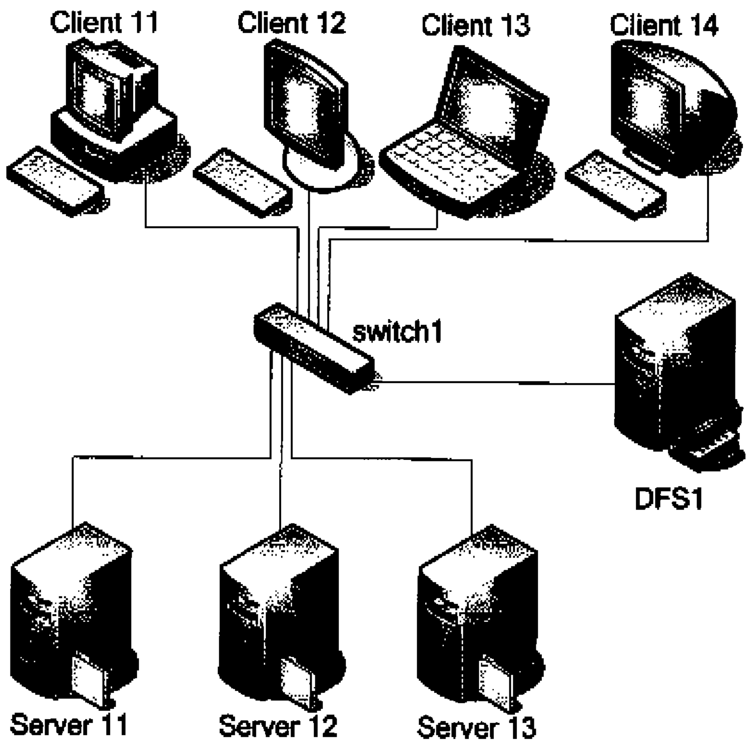

Client 11
Client 12
Client 13
Client 14
switch1
DFS1
Server 11
Server 12
Server 13

FIG. C-1

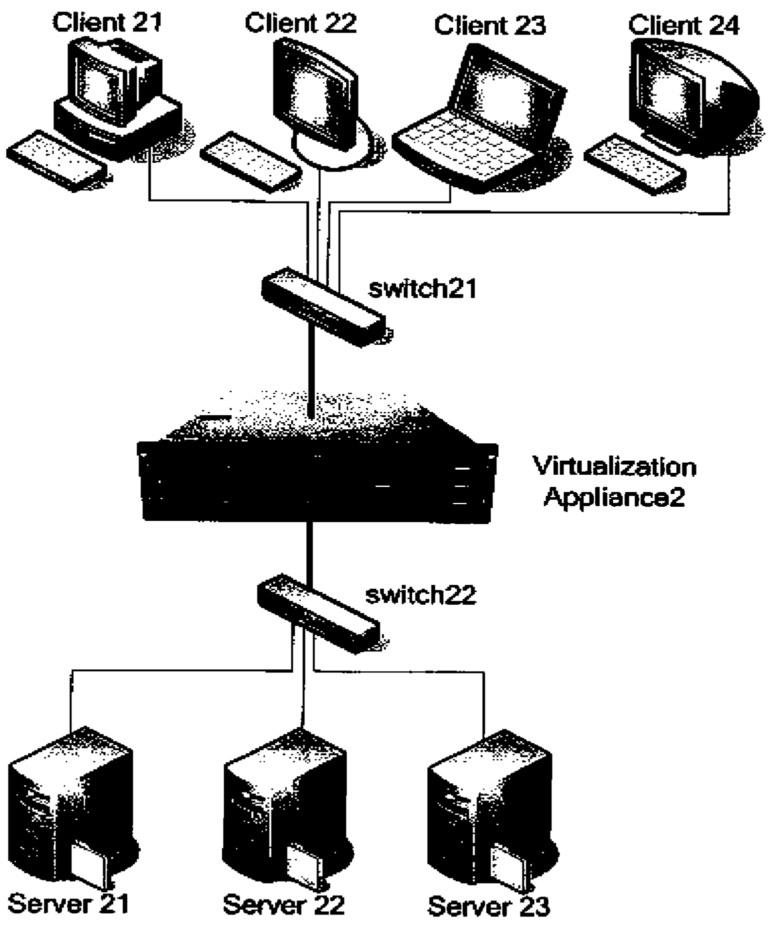

Client 21
Client 22
Client 23
Client 24
switch21
Virtualization Appliance2
switch22
Server 21
Server 22
Server 23

FIG. C-2

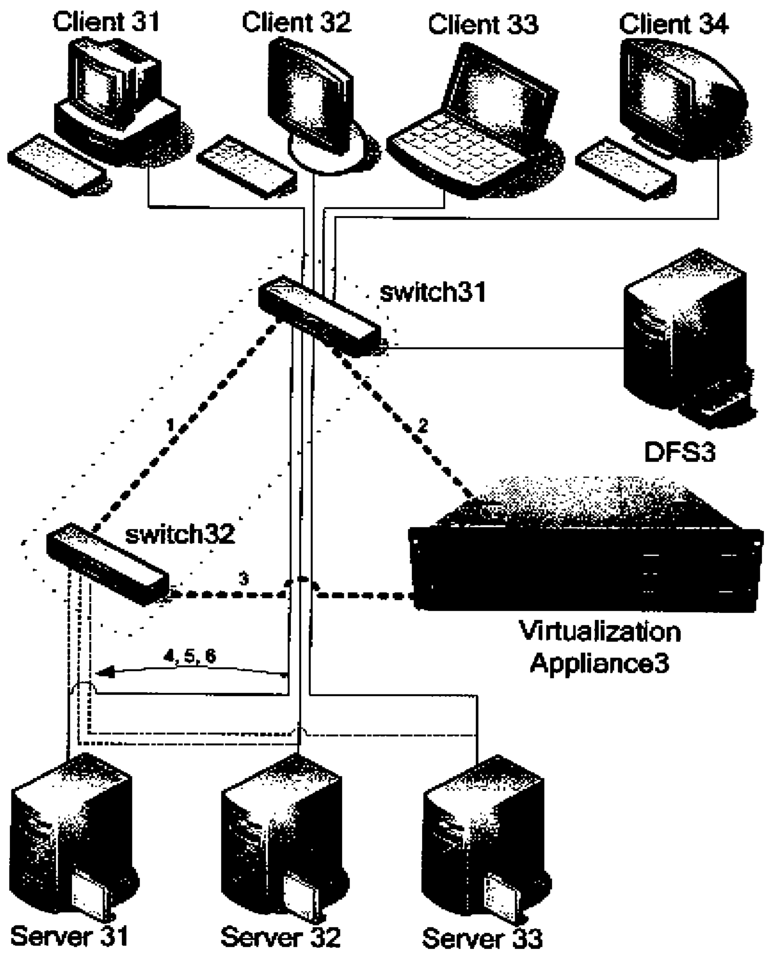

Client 31
Client 32
Client 33
Client 34
switch31
DFS3
1
2
switch32
3
Virtualization Appliance3
4, 5, 6
Server 31
Server 32
Server 33

FIG. C-3

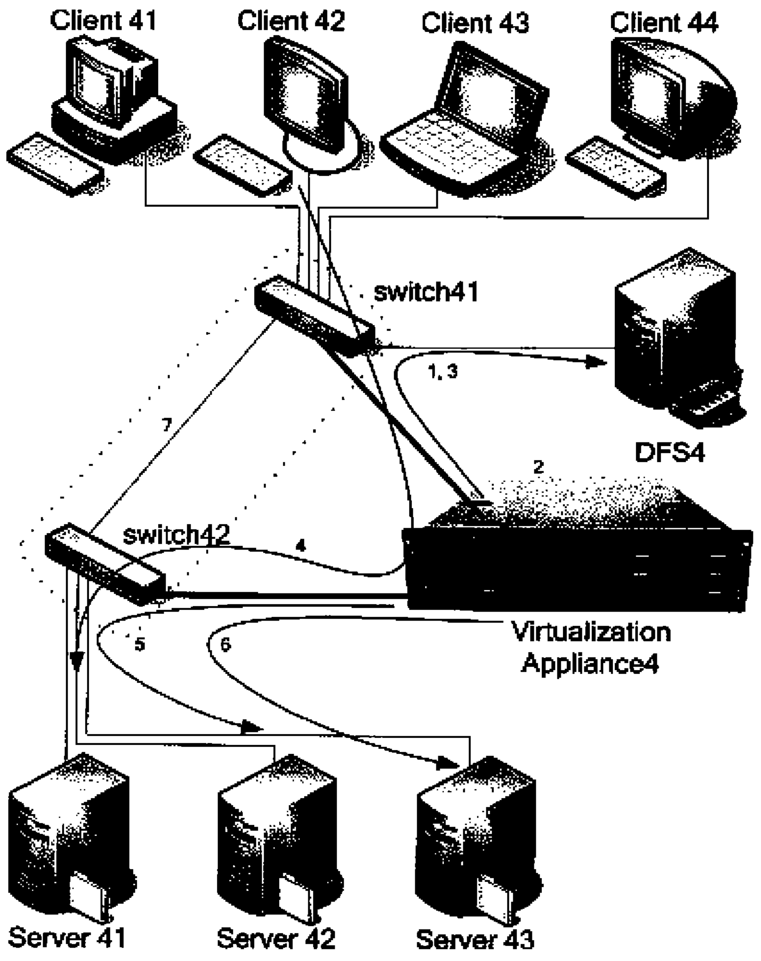

Client 41
Client 42
Client 43
Client 44
switch41
DFS4
1, 3
2
7
switch42
4
Virtualization Appliance4
5
6
Server 41
Server 42
Server 43

FIG. C-4

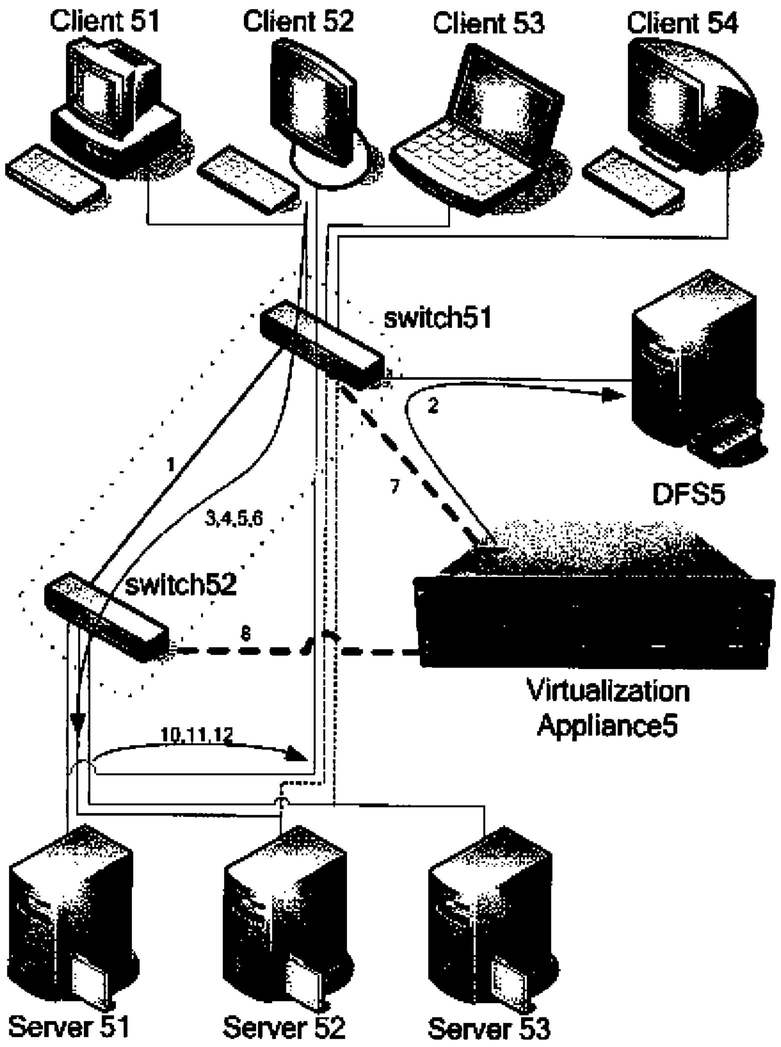

Client 51
Client 52
Client 53
Client 54
switch51
DFS5
2
1
7
3,4,5,6
switch52
8
Virtualization Appliance5
10,11,12
Server 51
Server 52
Server 53

FIG. C-5

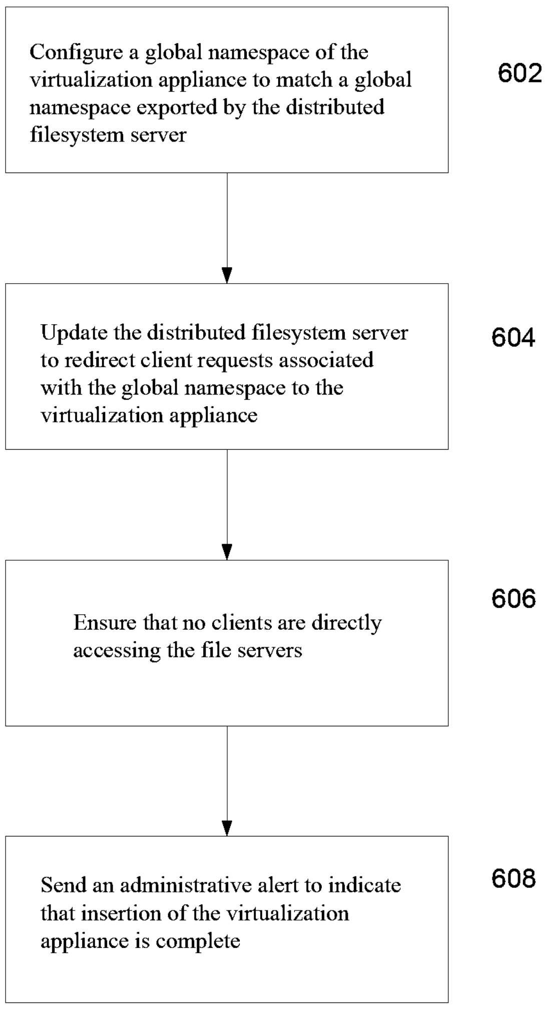

Configure a global namespace of the virtualization appliance to match a global namespace exported by the distributed filesystem server
602
Update the distributed filesystem server to redirect client requests associated with the global namespace to the virtualization appliance
604
Ensure that no clients are directly accessing the file servers
606
Send an administrative alert to indicate that insertion of the virtualization appliance is complete
608

FIG. C-6

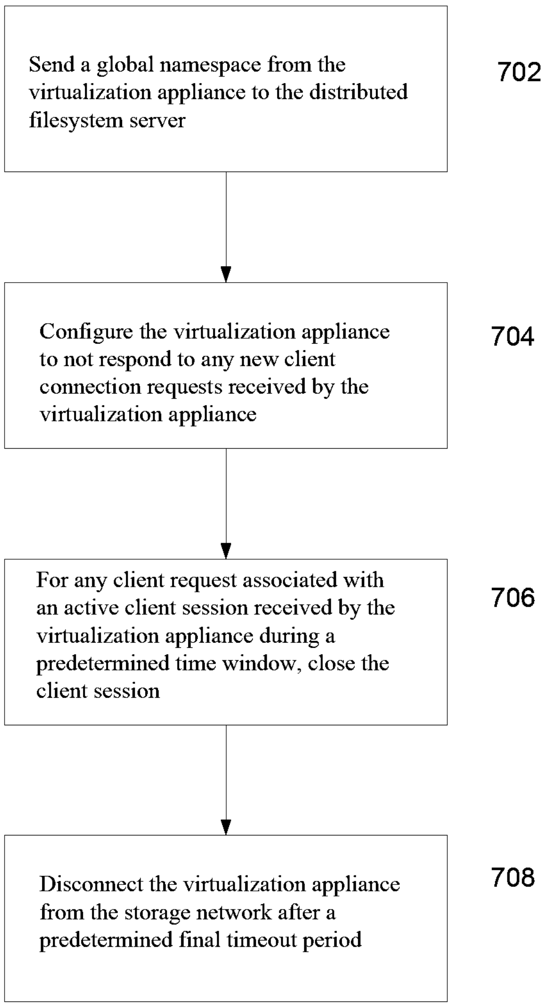

Send a global namespace from the virtualization appliance to the distributed filesystem server
702
Configure the virtualization appliance to not respond to any new client connection requests received by the virtualization appliance
704
For any client request associated with an active client session received by the virtualization appliance during a predetermined time window, close the client session
706
Disconnect the virtualization appliance from the storage network after a predetermined final timeout period
708

FIG. C-7

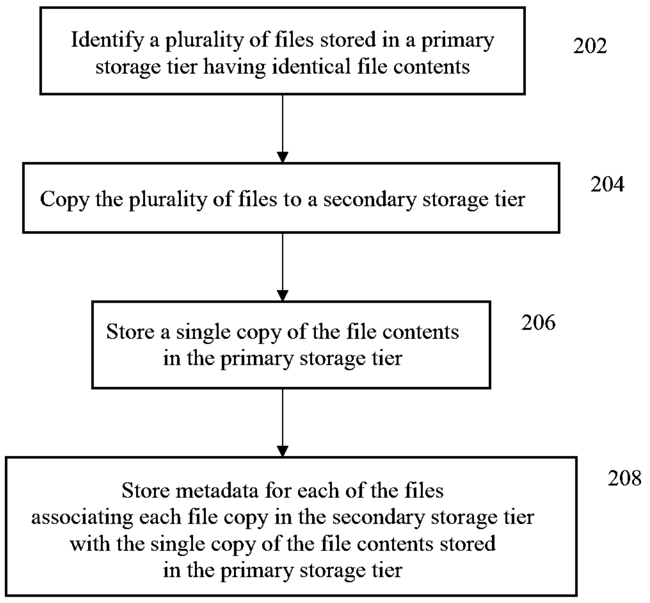

Identify a plurality of files stored in a primary storage tier having identical file contents
202
Copy the plurality of files to a secondary storage tier
204
Store a single copy of the file contents in the primary storage tier
206
Store metadata for each of the files associating each file copy in the secondary storage tier with the single copy of the file contents stored in the primary storage tier
208

FIG. D-1

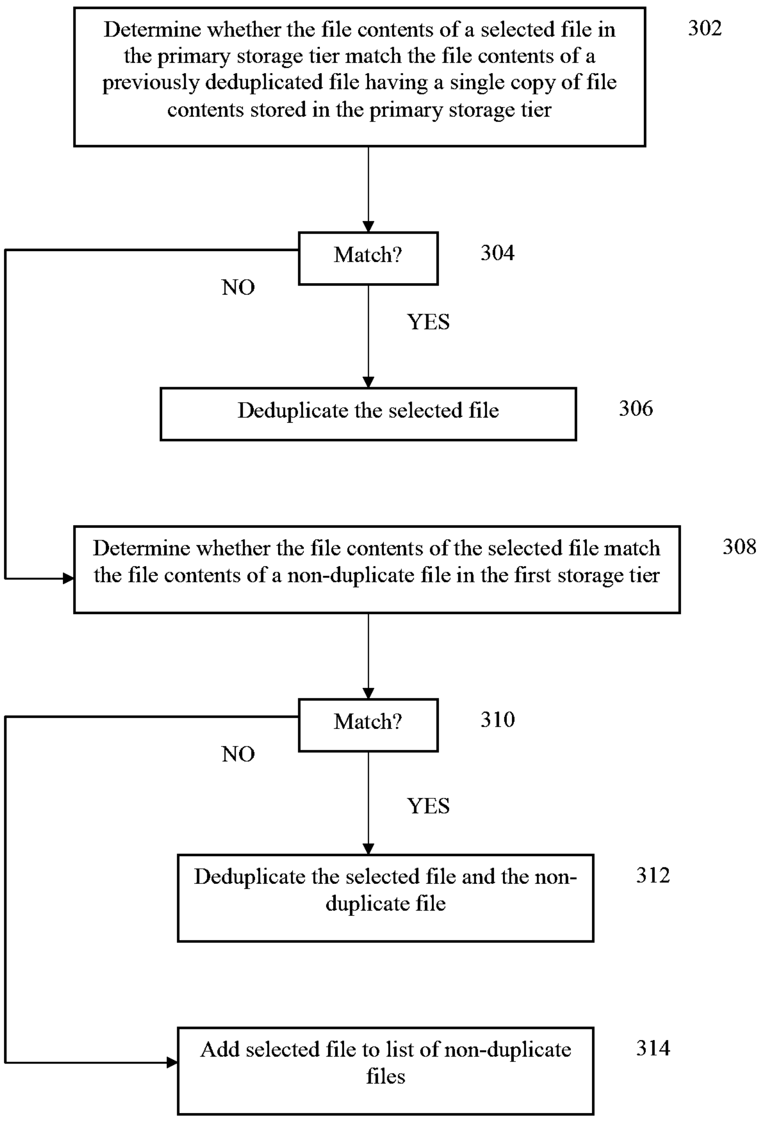

Determine whether the file contents of a selected file in the primary storage tier match the file contents of a previously deduplicated file having a single copy of file contents stored in the primary storage tier
302
Match?
304
NO
YES
Deduplicate the selected file
306
Determine whether the file contents of the selected file match the file contents of a non-duplicate file in the first storage tier
308
Match?
310
NO
YES
Deduplicate the selected file and the non-duplicate file
312
Add selected file to list of non-duplicate files
314

FIG. D-2

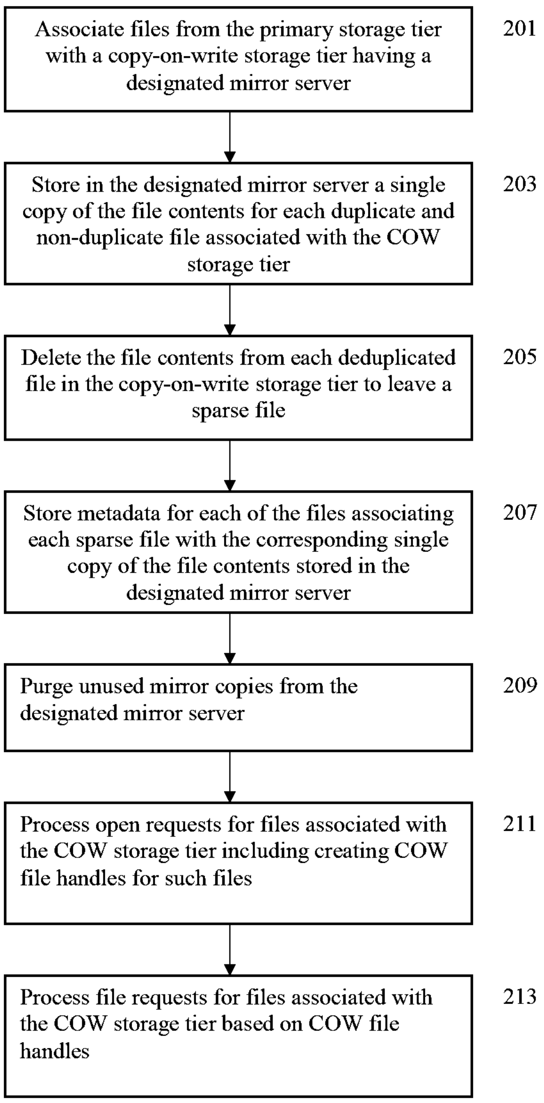

Associate files from the primary storage tier with a copy-on-write storage tier having a designated mirror server
201
Store in the designated mirror server a single copy of the file contents for each duplicate and non-duplicate file associated with the COW storage tier
203
Delete the file contents from each deduplicated file in the copy-on-write storage tier to leave a sparse file
205
Store metadata for each of the files associating each sparse file with the corresponding single copy of the file contents stored in the designated mirror server
207
Purge unused mirror copies from the designated mirror server
209
Process open requests for files associated with the COW storage tier including creating COW file handles for such files
211
Process file requests for files associated with the COW storage tier based on COW file handles
213

FIG. E-1