US008627446B1

(12) **United States Patent**
Eaton et al.

(10) **Patent No.: US 8,627,446 B1**
(45) **Date of Patent: Jan. 7, 2014**

(54) **FEDERATING DATA BETWEEN GROUPS OF SERVERS**

(75) Inventors: **Patrick R. Eaton**, Arlington, MA (US); **Zane Pan**, Lexington, MA (US); **Rossen Dimitrov**, Nashua, NH (US)

(73) Assignee: **EMC Corporation**, Hopkinton, MA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 499 days.

(21) Appl. No.: **12/586,939**

(22) Filed: **Sep. 30, 2009**

(51) **Int. Cl.**
H04L 29/06 (2006.01)
G06F 21/00 (2013.01)

(52) **U.S. Cl.**
USPC **726/12**

(58) **Field of Classification Search**
None
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

7,412,492	B1	8/2008	Waldspurger	
7,433,951	B1	10/2008	Waldspurger	
7,577,722	B1	8/2009	Khandekar et al.	
8,060,926	B1 *	11/2011	Ebrahimi et al.	726/12
2002/0152318	A1 *	10/2002	Menon et al.	709/231
2003/0097443	A1 *	5/2003	Gillett et al.	709/225
2005/0223412	A1 *	10/2005	Nadalin et al.	726/3
2006/0212935	A1 *	9/2006	Burman et al.	726/12
2008/0091812	A1 *	4/2008	Lev-Ran et al.	709/223
2008/0126547	A1	5/2008	Waldspurger	
2009/0112789	A1	4/2009	Oliveira et al.	
2009/0112811	A1	4/2009	Oliveira et al.	
2009/0112879	A1	4/2009	Oliveira et al.	
2009/0112880	A1	4/2009	Oliveira et al.	
2009/0112921	A1	4/2009	Oliveira et al.	

OTHER PUBLICATIONS

J. Matthews et al., "Virtual Machine Contracts for Datacenter and Cloud Computing Environments," *ACDC* '09, Jun. 19, 2009, Barcelona, Spain, 6 pp.
W. Zheng et al., "Cloud Service and Service Selection Algorithm Research," *GEC* '09, Jun. 12-14, 2009, Shanghai, China, pp. 1045-1048.
"Architectural Strategies for Cloud Computing," Oracle White Paper in Enterprise Architecture, Aug. 2009, 18 pp.
"VMware Virtual Machine File System: Technical Overview and Best Practices," VMware Technical White Paper, Version 1.0, WP-022-PRD-01-01, 2007, 19 pp.
"IT in the Cloud: Using VMware vCloud for Reliable, Flexible, Shared IT Resources," VMware White Paper, 2008, 8 pp.
"VMware vSphere, the First Cloud Operating System, Provides an Evolutionary, Non-disruptive Path to Cloud Computing," VMware White Paper, 2009, 8 pp.
"VMware vSphere 4," VMware Cheat Sheet, 2009, 9 pp.
"Amazon Elastic Compute Cloud: Developer Guide," API Vers. Jul. 15, 2009, Amazon web services, 2009, 154 pp.
"EMC Atmos Online Compute Service: Getting Started Guide," Document Revision A, Sep. 28, 2009, 35 pp.

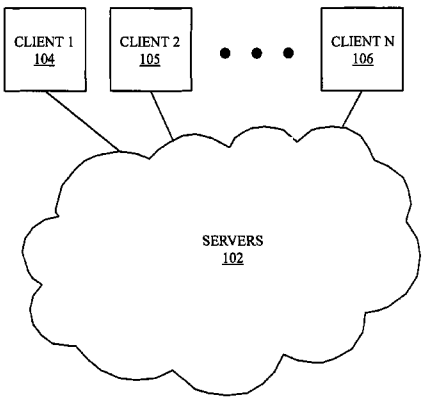
* cited by examiner

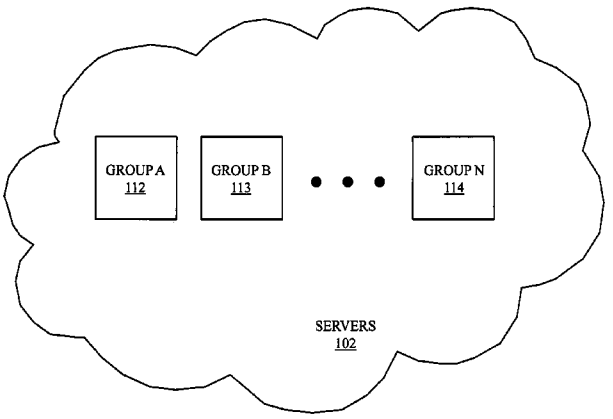
Primary Examiner — Gilberto Barron, Jr.
Assistant Examiner — Malcolm Cribbs
(74) *Attorney, Agent, or Firm* — Muirhead and Saturnelli, LLC

(57) **ABSTRACT**

Accessing data includes determining if the data is provided on a local group of servers or on an external group of servers. If the data is provided on a local group of servers, a storage server is used to access the data. If the data is provided on an external group of servers, a proxy server is used to access the data. The proxy server interacts with an entity accessing the data in a manner that is substantially similar to interaction between the entity and the storage server. Using the proxy server may include initially providing an account id and a password. Following providing an account id and a password, using the proxy server may include using an account id and a shared secret. Using the proxy server may include using RSA ID tokens or cryptographic certificates.

17 Claims, 26 Drawing Sheets





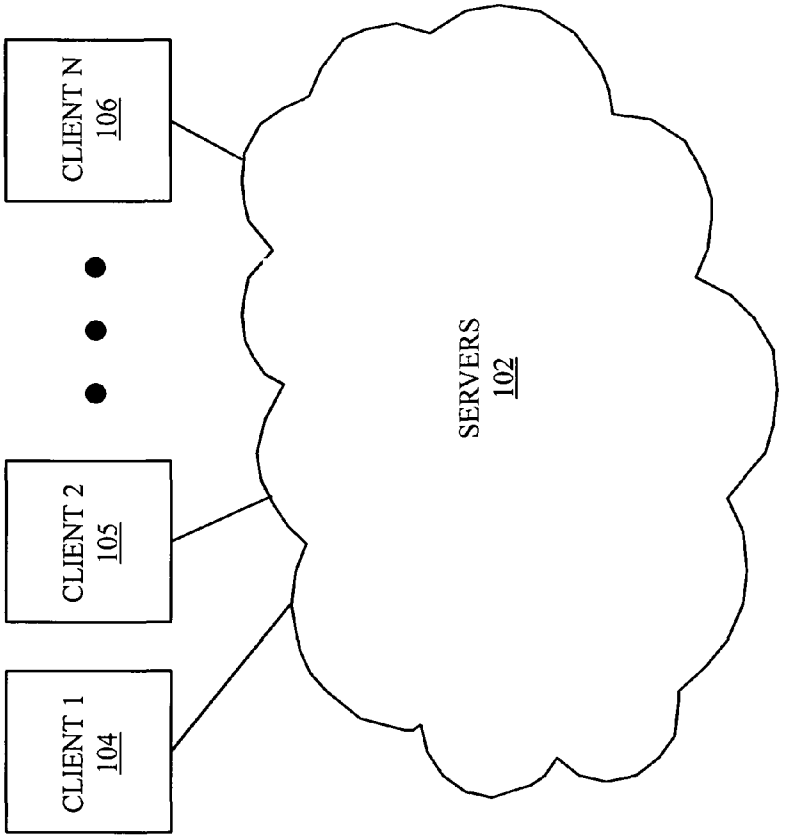


FIGURE 1A

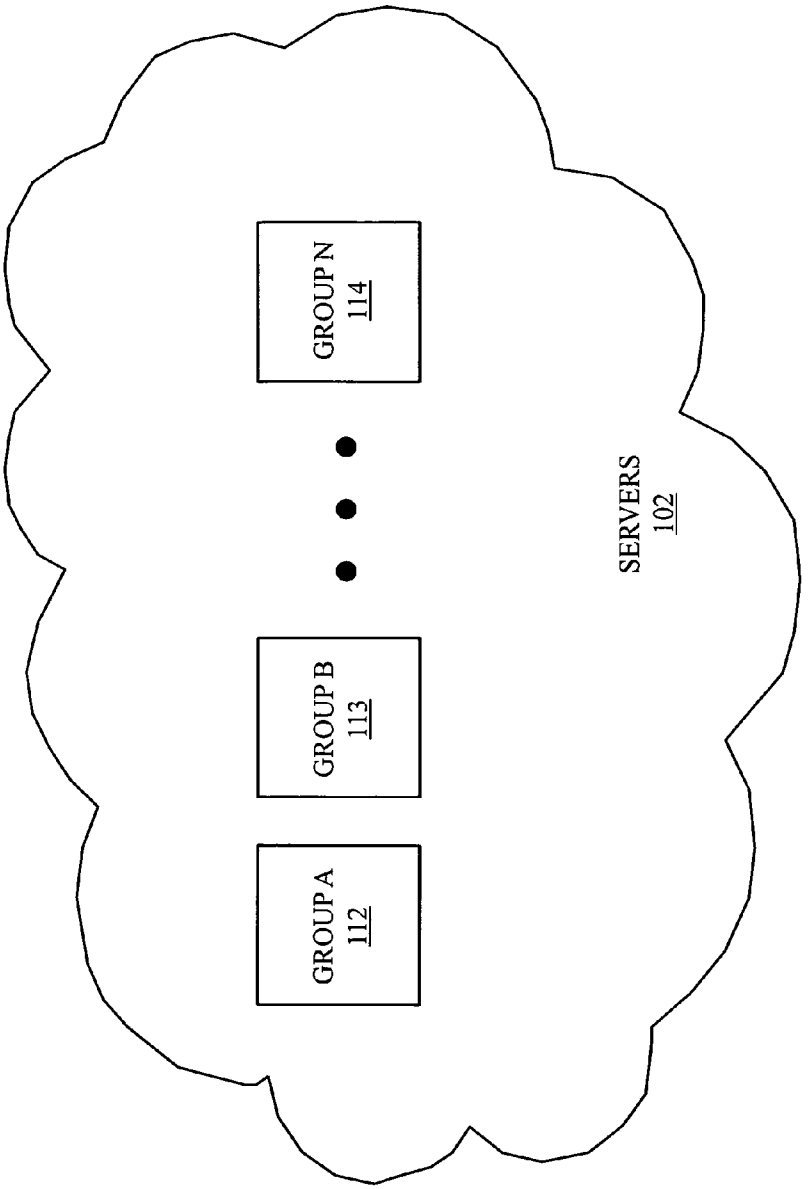


FIGURE 1B

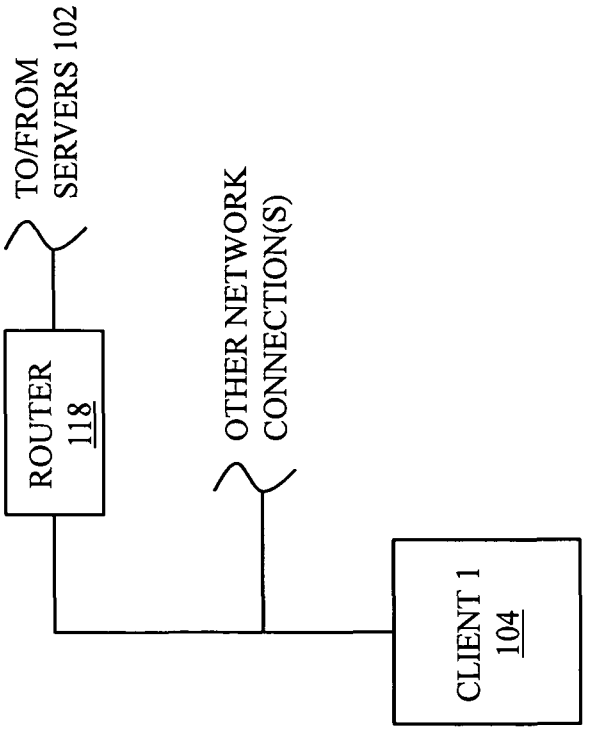


FIGURE 2B

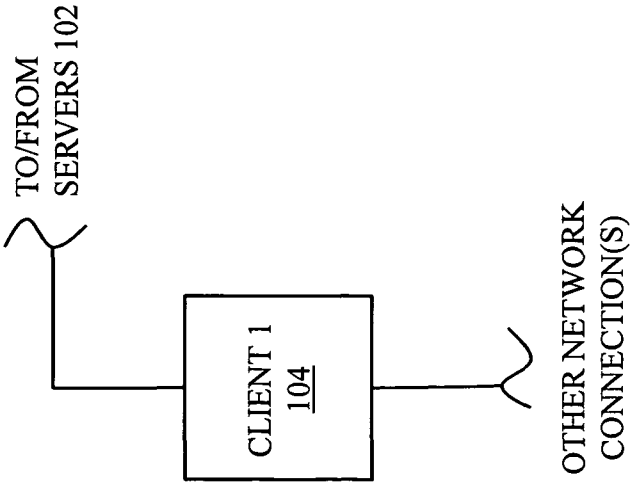


FIGURE 2A

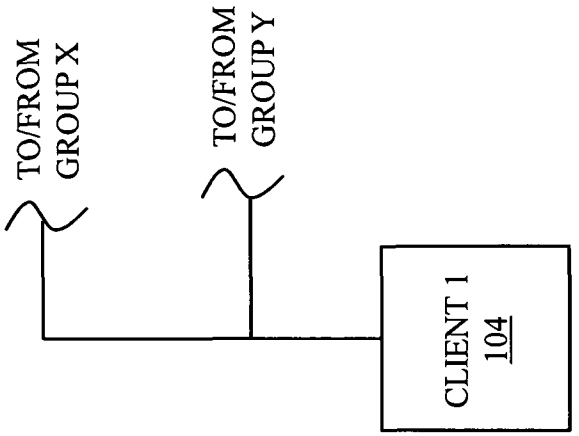


FIGURE 2D

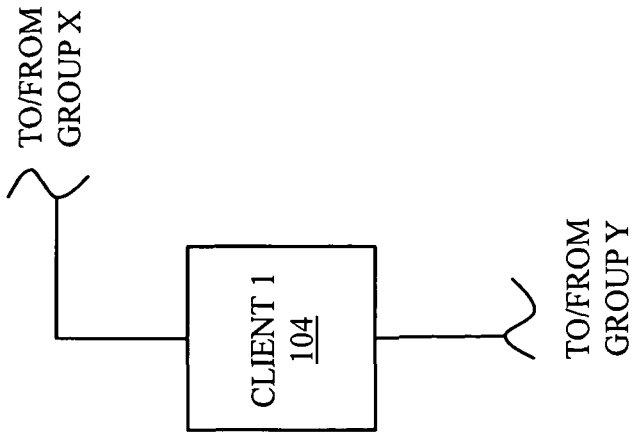


FIGURE 2C

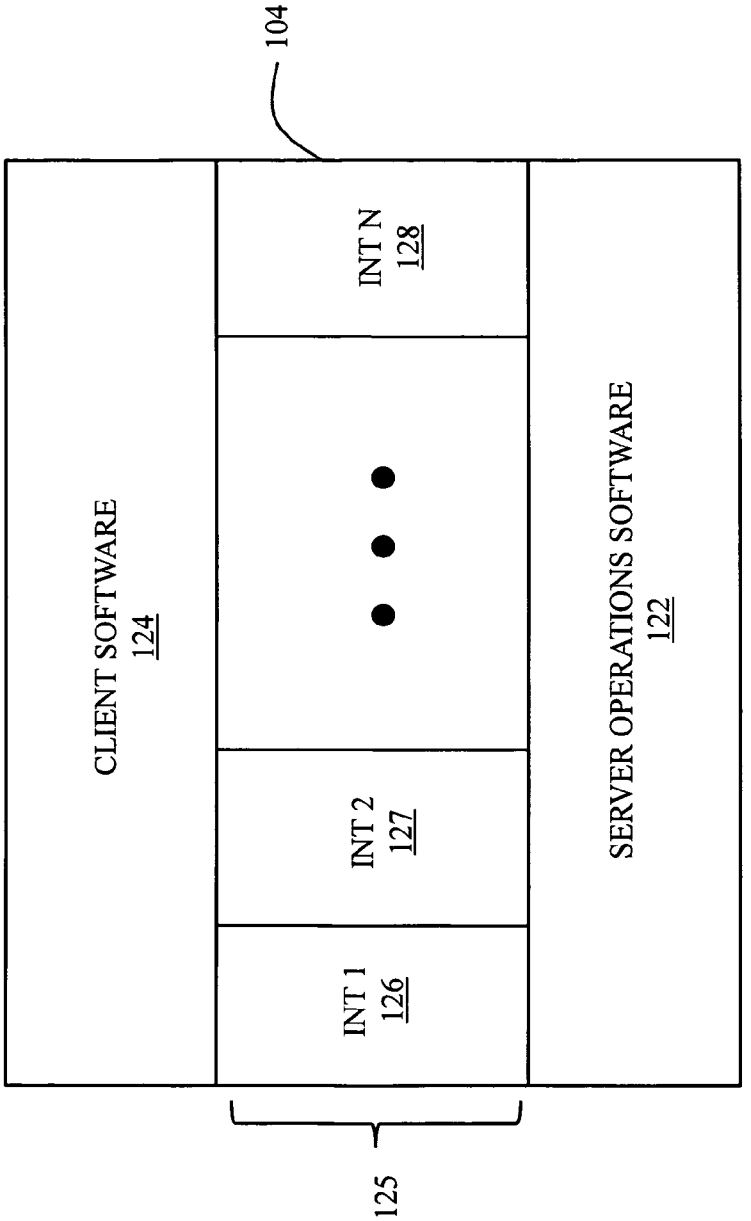

104
CLIENT SOFTWARE
124
INT 1
126
INT 2
127
INT N
128
SERVER OPERATIONS SOFTWARE
122
125

FIGURE 3

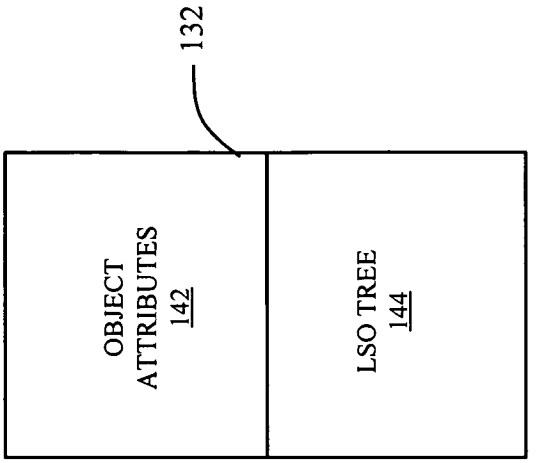


FIGURE 5

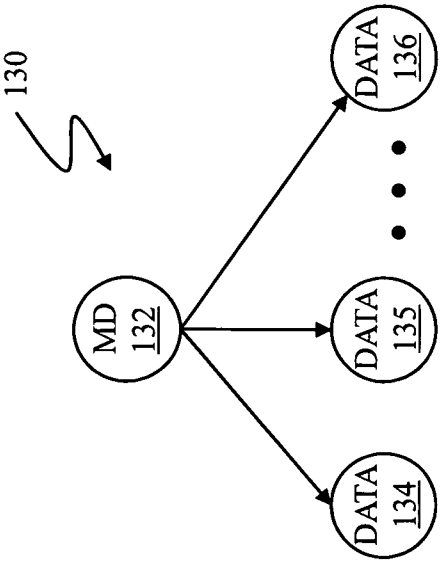


FIGURE 4

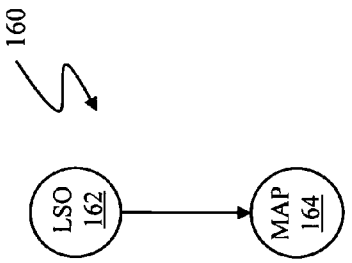


FIGURE 6A

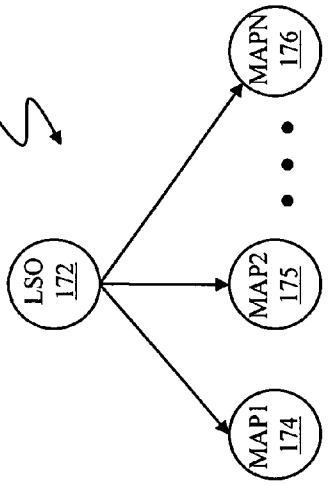


FIGURE 6B

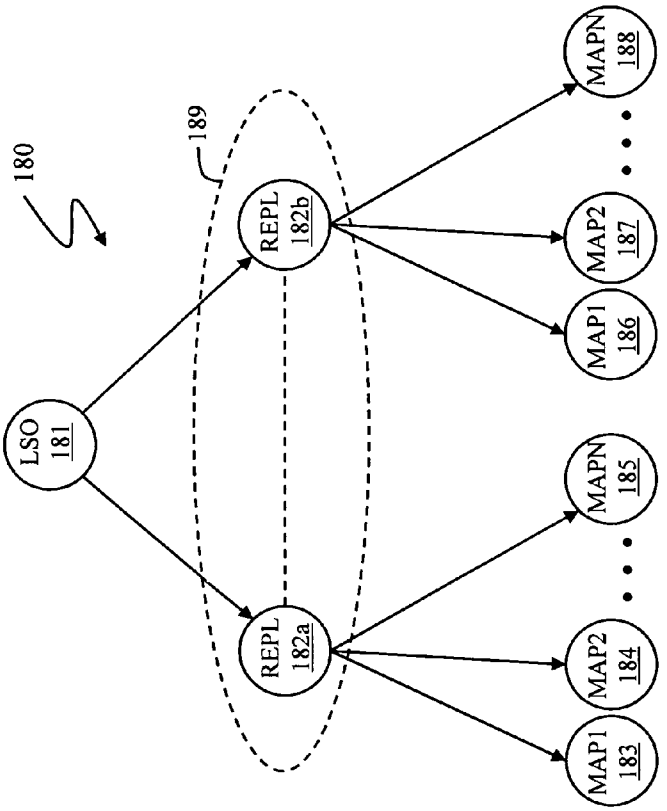


FIGURE 6C

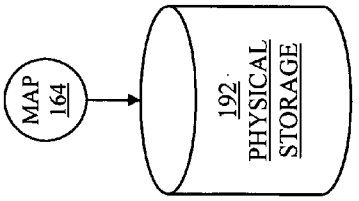


FIGURE 7A

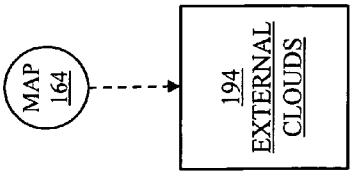


FIGURE 7B

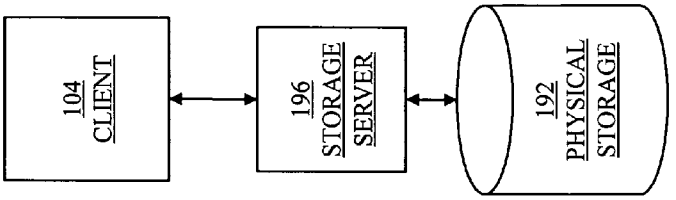


FIGURE 7C

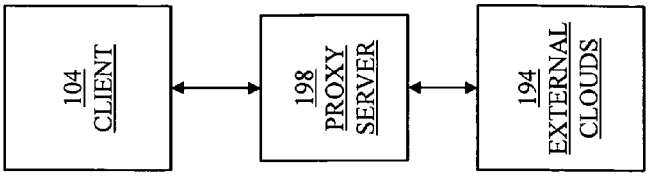


FIGURE 7D

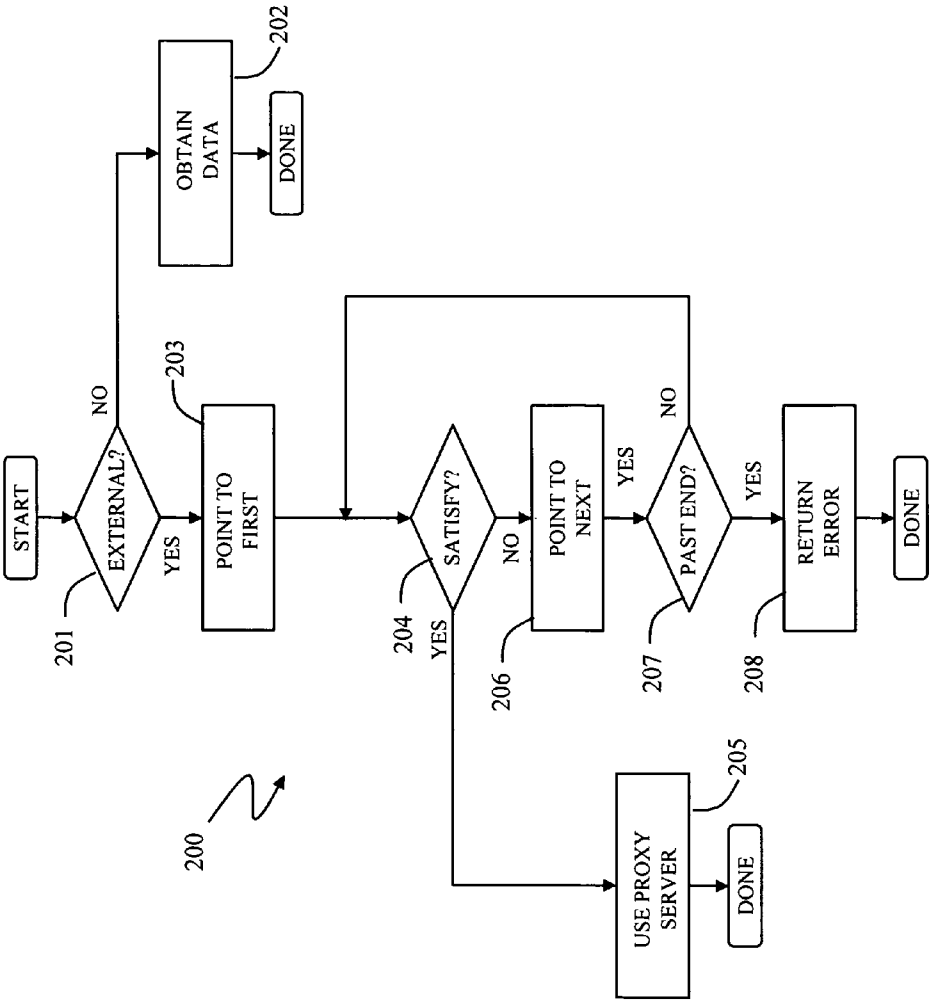

START
EXTERNAL?
NO
YES
OBTAIN DATA
DONE
POINT TO FIRST
SATISFY?
YES
NO
USE PROXY SERVER
DONE
POINT TO NEXT
PAST END?
NO
YES
RETURN ERROR
DONE
200
201
202
203
204
205
206
207
208

FIGURE 8

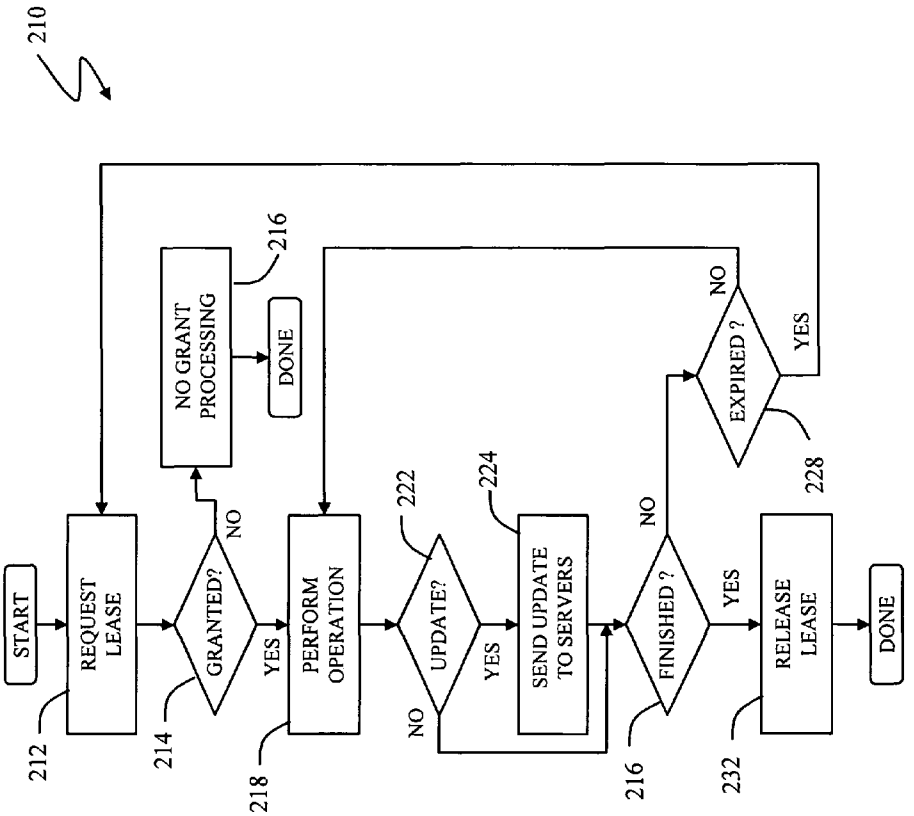

210
START
REQUEST LEASE
212
GRANTED?
214
NO
NO GRANT PROCESSING
216
DONE
YES
PERFORM OPERATION
218
UPDATE?
222
NO
YES
SEND UPDATE TO SERVERS
224
FINISHED ?
216
NO
EXPIRED ?
228
NO
YES
YES
RELEASE LEASE
232
DONE

FIGURE 9

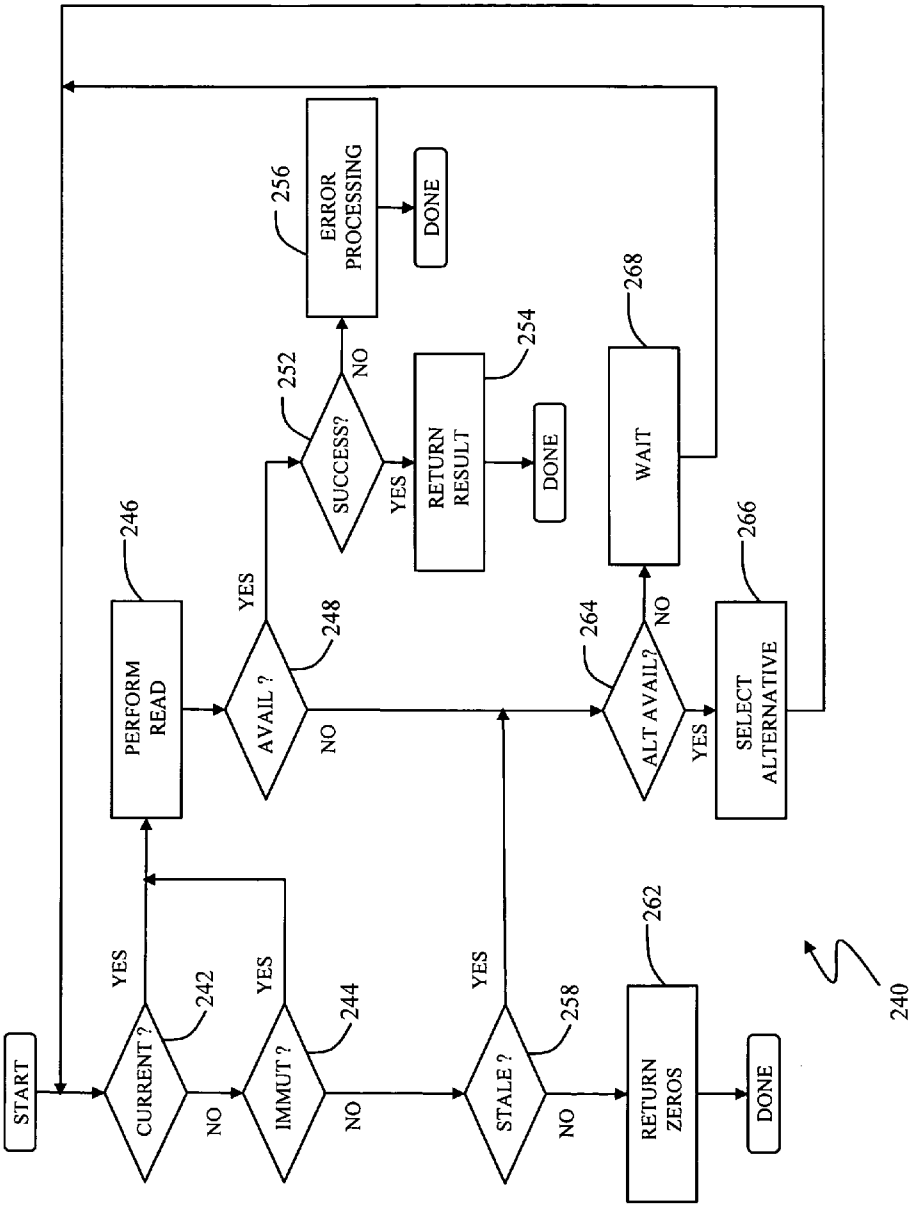

START
CURRENT ?
242
YES
NO
IMMUT ?
244
YES
NO
STALE ?
258
YES
NO
RETURN ZEROS
262
DONE
PERFORM READ
246
AVAIL ?
248
YES
NO
SUCCESS?
252
YES
NO
RETURN RESULT
254
DONE
ERROR PROCESSING
256
DONE
ALT AVAIL?
264
YES
NO
WAIT
268
SELECT ALTERNATIVE
266
240

FIGURE 10

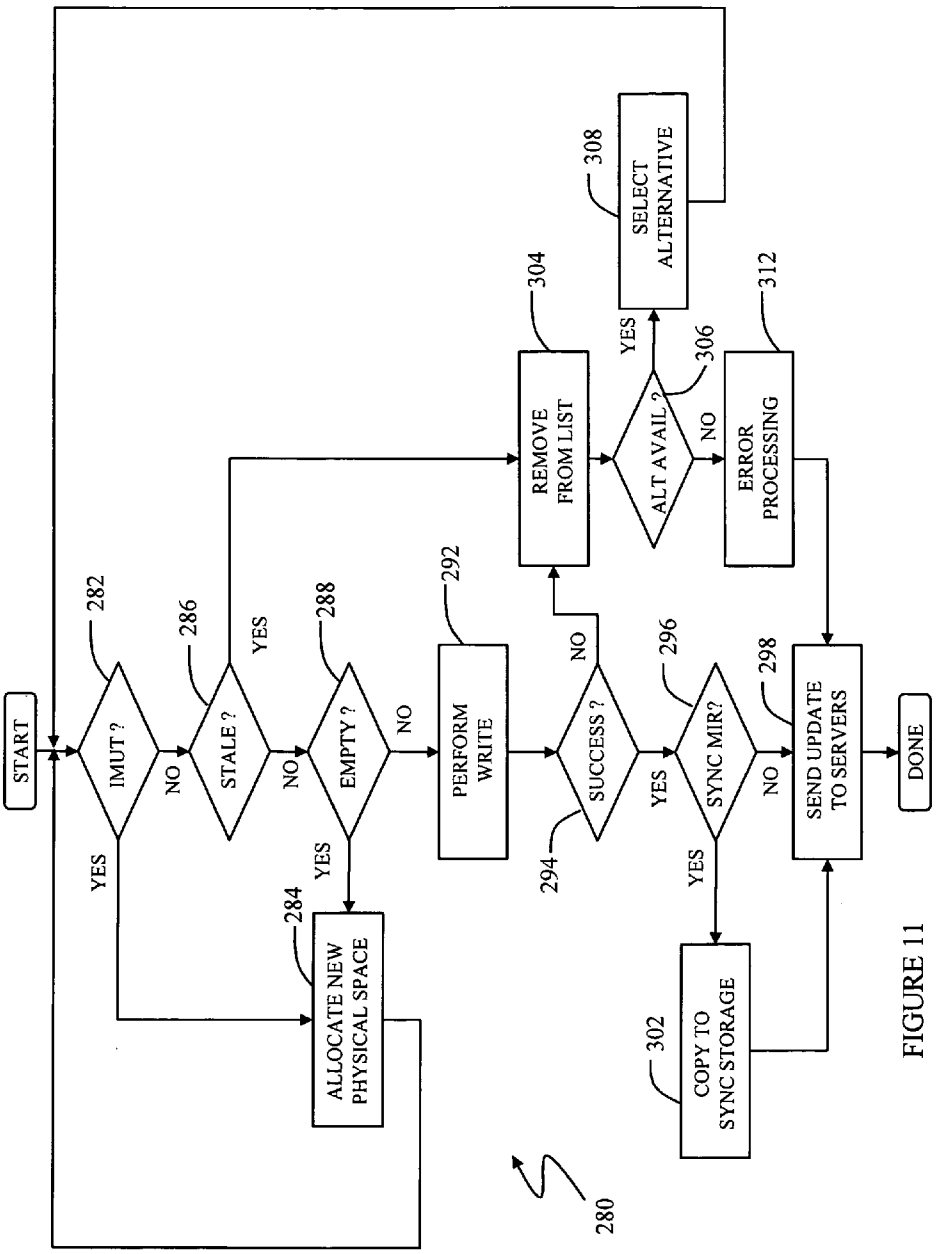

START
IMUT ?
282
YES
NO
STALE ?
286
YES
NO
EMPTY ?
288
YES
NO
ALLOCATE NEW PHYSICAL SPACE
284
PERFORM WRITE
292
SUCCESS ?
294
NO
YES
REMOVE FROM LIST
304
ALT AVAIL ?
306
YES
NO
SELECT ALTERNATIVE
308
ERROR PROCESSING
312
SYNC MIR?
296
YES
NO
COPY TO SYNC STORAGE
302
SEND UPDATE TO SERVERS
298
DONE
280

FIGURE 11

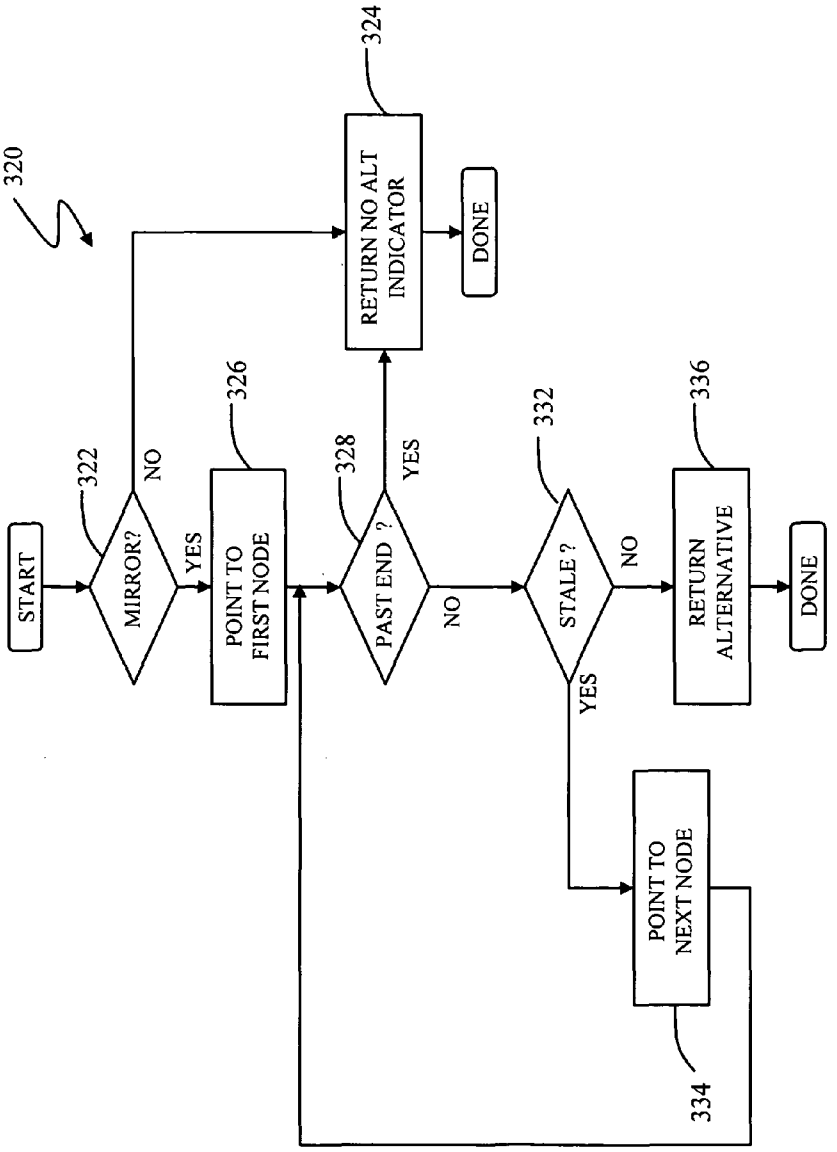

320
START
322
MIRROR?
NO
YES
326
POINT TO FIRST NODE
328
PAST END ?
YES
NO
324
RETURN NO ALT INDICATOR
DONE
332
STALE ?
NO
YES
336
RETURN ALTERNATIVE
DONE
334
POINT TO NEXT NODE

FIGURE 12

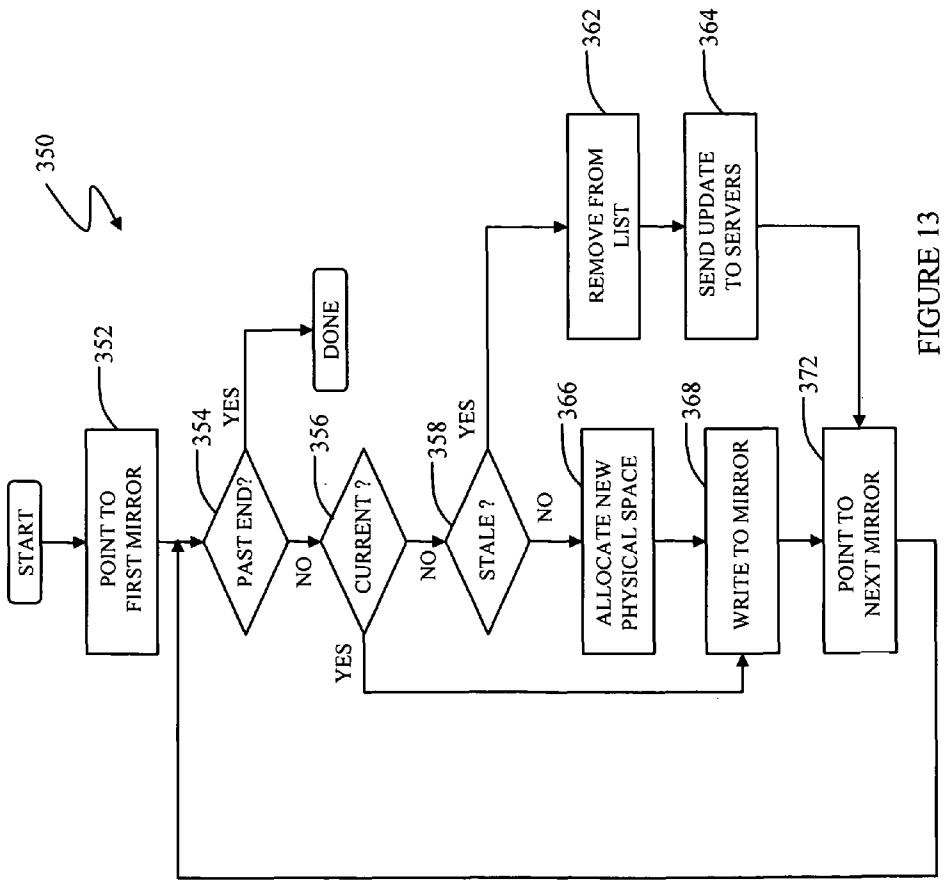

350
START
352
POINT TO FIRST MIRROR
354
PAST END?
YES
DONE
NO
356
CURRENT ?
YES
NO
358
STALE ?
YES
NO
362
REMOVE FROM LIST
364
SEND UPDATE TO SERVERS
366
ALLOCATE NEW PHYSICAL SPACE
368
WRITE TO MIRROR
372
POINT TO NEXT MIRROR

FIGURE 13

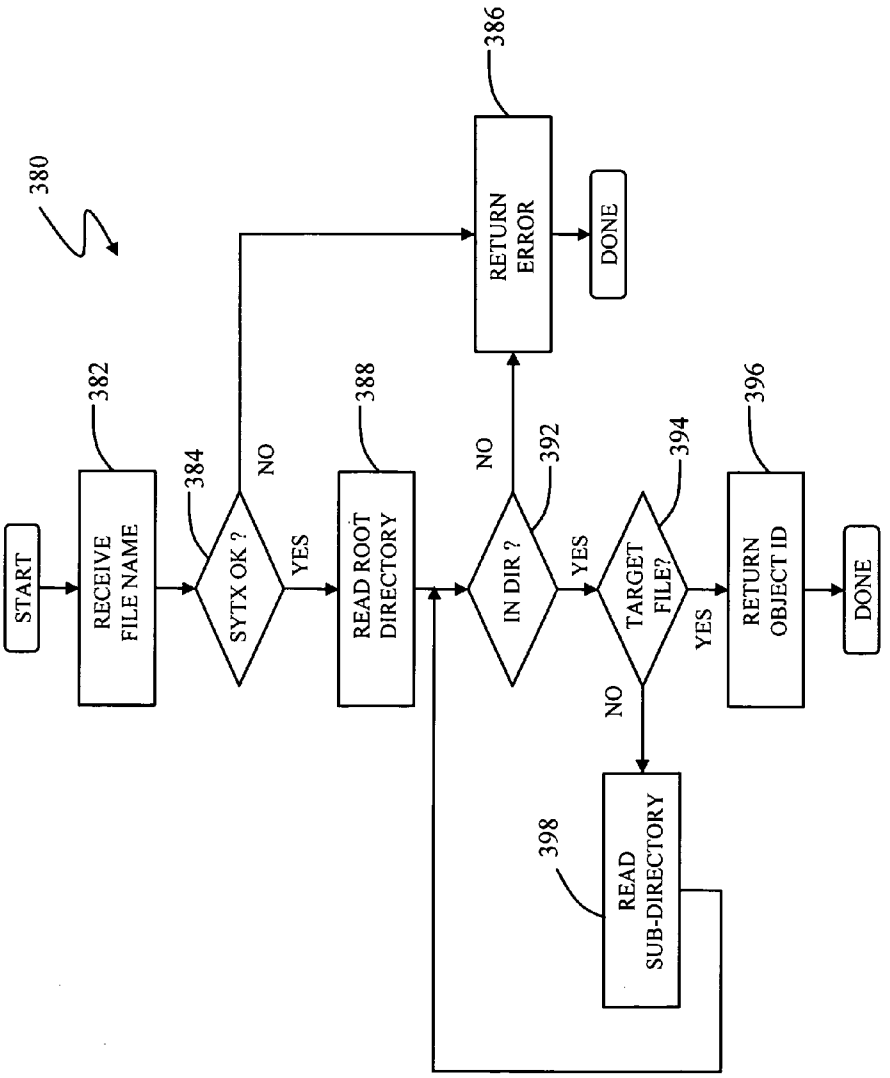

380
START
382
RECEIVE FILE NAME
384
SYTX OK ?
NO
YES
386
RETURN ERROR
DONE
388
READ ROOT DIRECTORY
IN DIR ?
NO
392
YES
394
TARGET FILE?
NO
YES
396
RETURN OBJECT ID
DONE
398
READ SUB-DIRECTORY

FIGURE 14

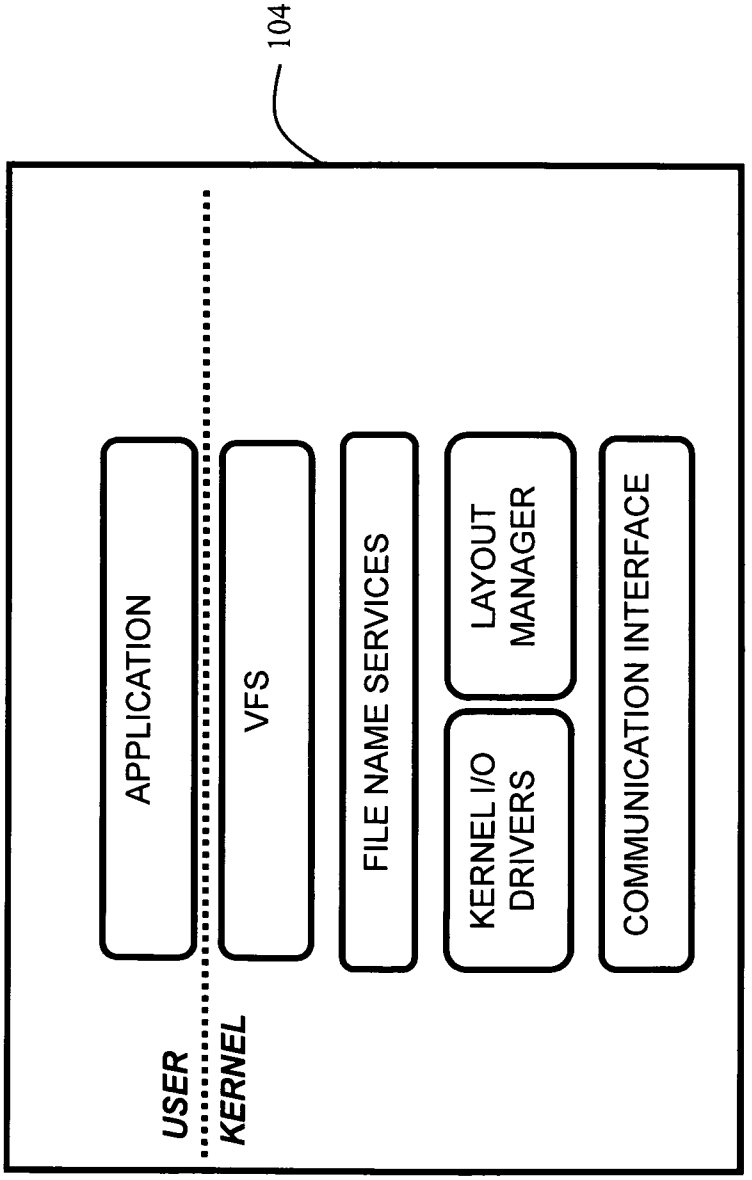

104
USER
KERNEL
APPLICATION
VFS
FILE NAME SERVICES
KERNEL I/O DRIVERS
LAYOUT MANAGER
COMMUNICATION INTERFACE

FIGURE 15

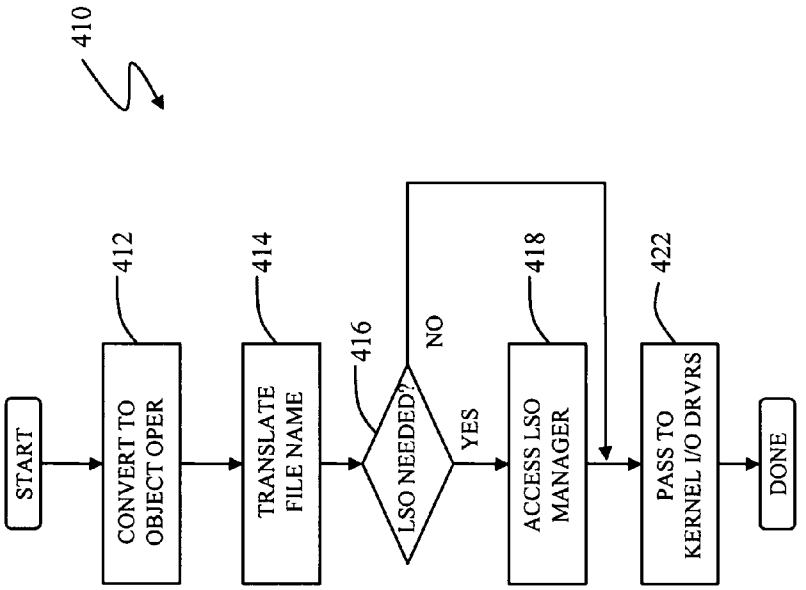

410
START
412
CONVERT TO OBJECT OPER
414
TRANSLATE FILE NAME
416
LSO NEEDED?
NO
YES
418
ACCESS LSO MANAGER
422
PASS TO KERNEL I/O DRVRS
DONE

FIGURE 16

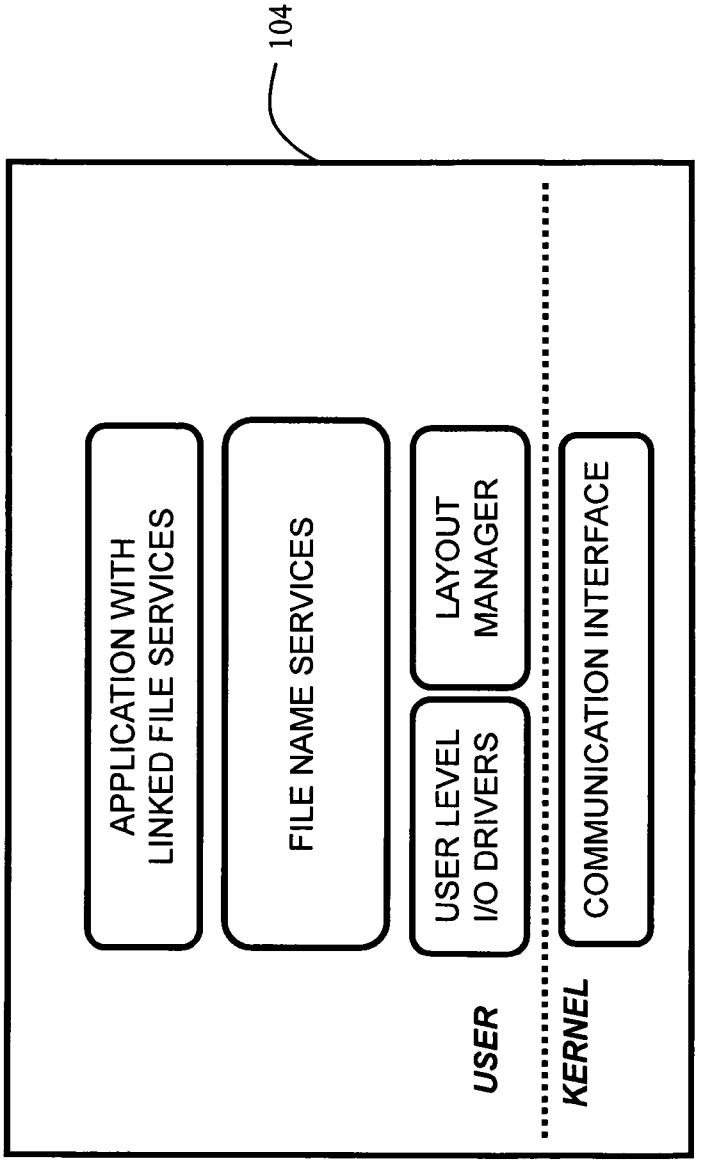

104
APPLICATION WITH LINKED FILE SERVICES
FILE NAME SERVICES
USER LEVEL I/O DRIVERS
LAYOUT MANAGER
COMMUNICATION INTERFACE
USER
KERNEL

FIGURE 17

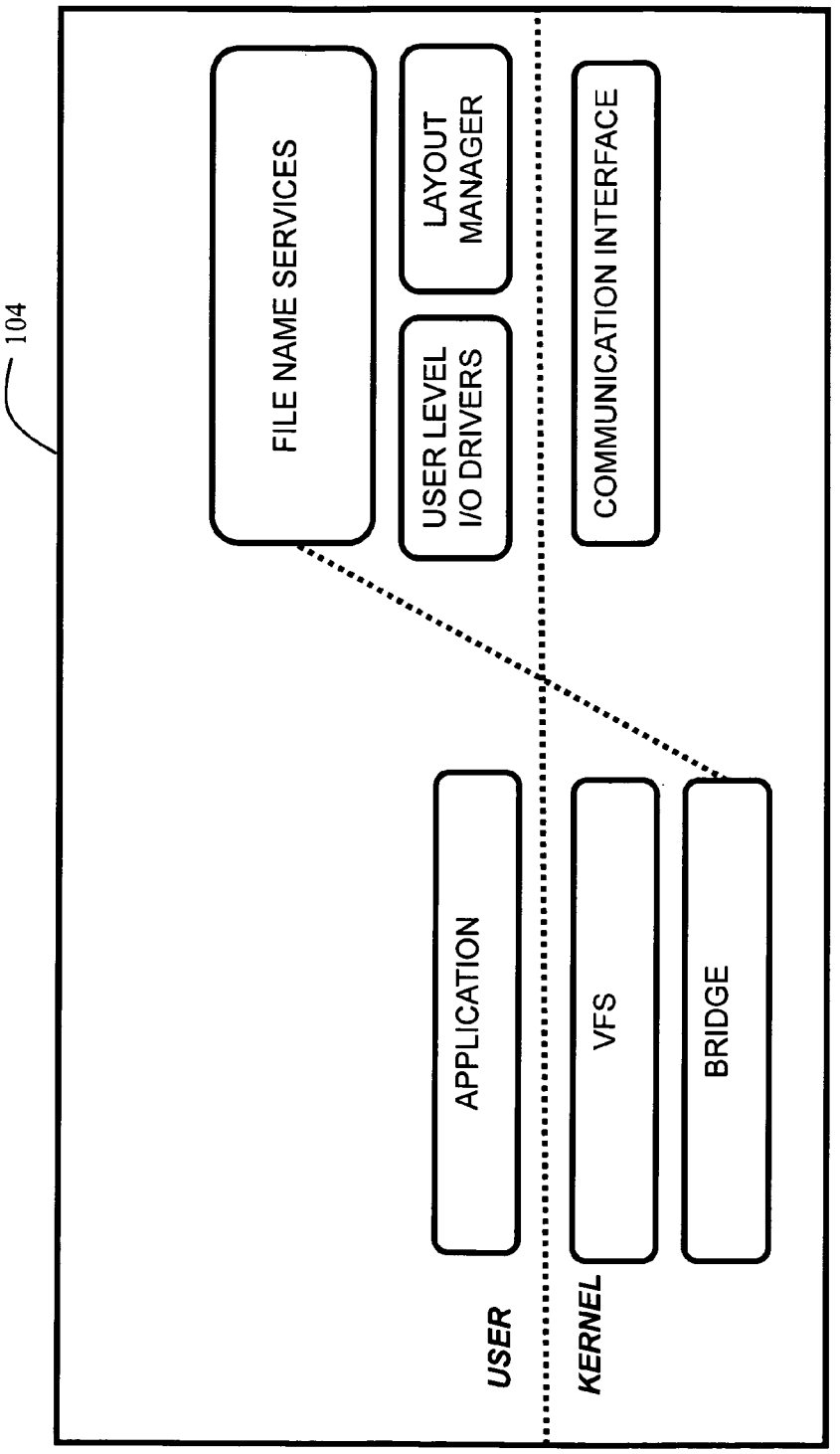

104
FILE NAME SERVICES
USER LEVEL I/O DRIVERS
LAYOUT MANAGER
COMMUNICATION INTERFACE
APPLICATION
VFS
BRIDGE
USER
KERNEL

FIGURE 18

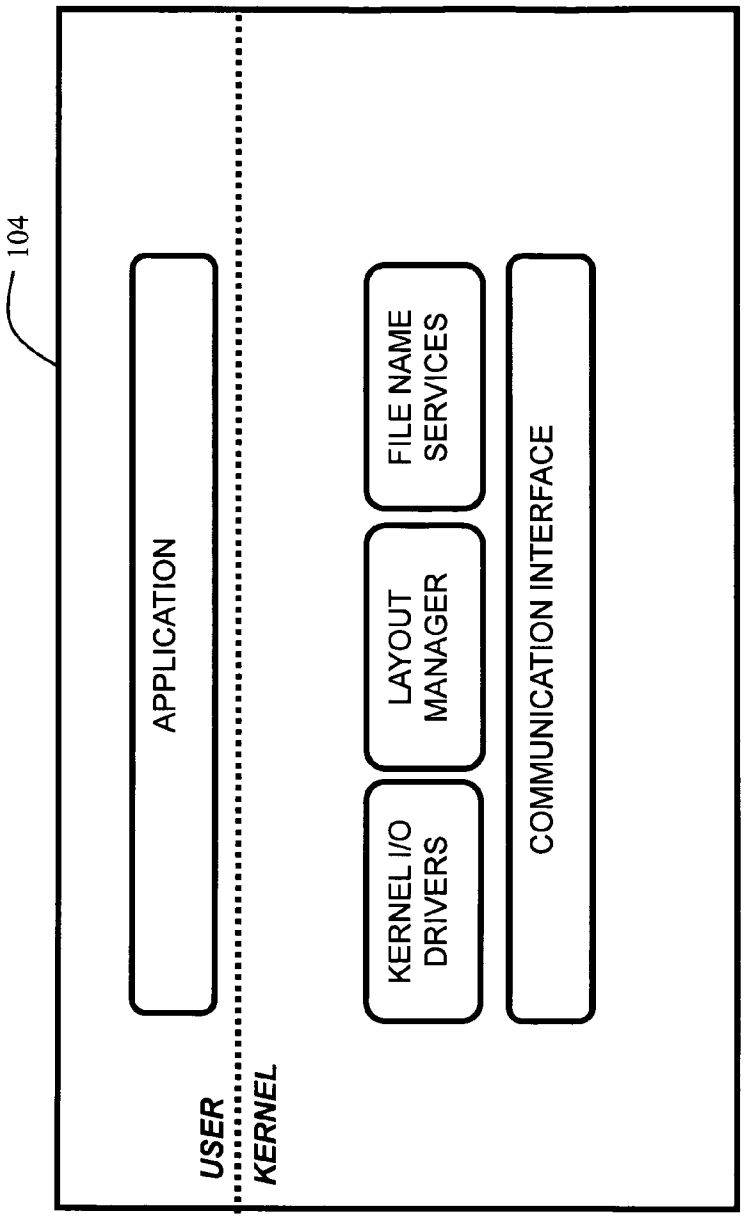

104
APPLICATION
USER
KERNEL
KERNEL I/O DRIVERS
LAYOUT MANAGER
FILE NAME SERVICES
COMMUNICATION INTERFACE

FIGURE 19

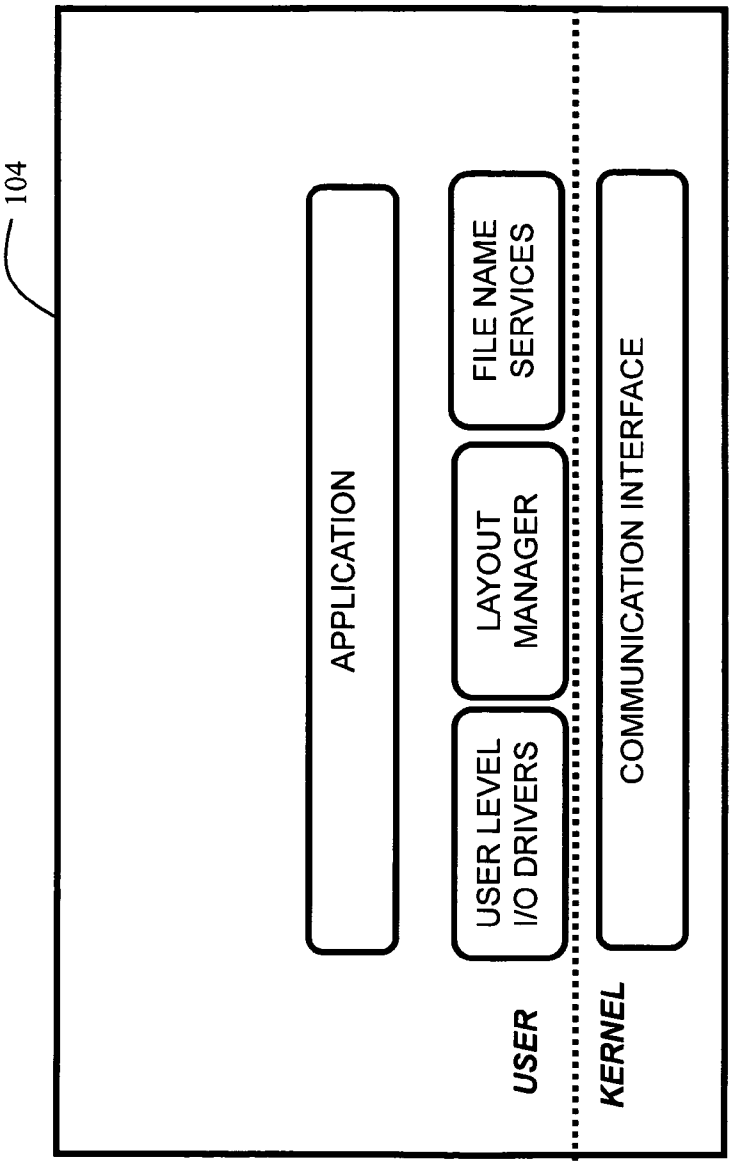

104
APPLICATION
USER LEVEL I/O DRIVERS
LAYOUT MANAGER
FILE NAME SERVICES
COMMUNICATION INTERFACE
USER
KERNEL

FIGURE 20

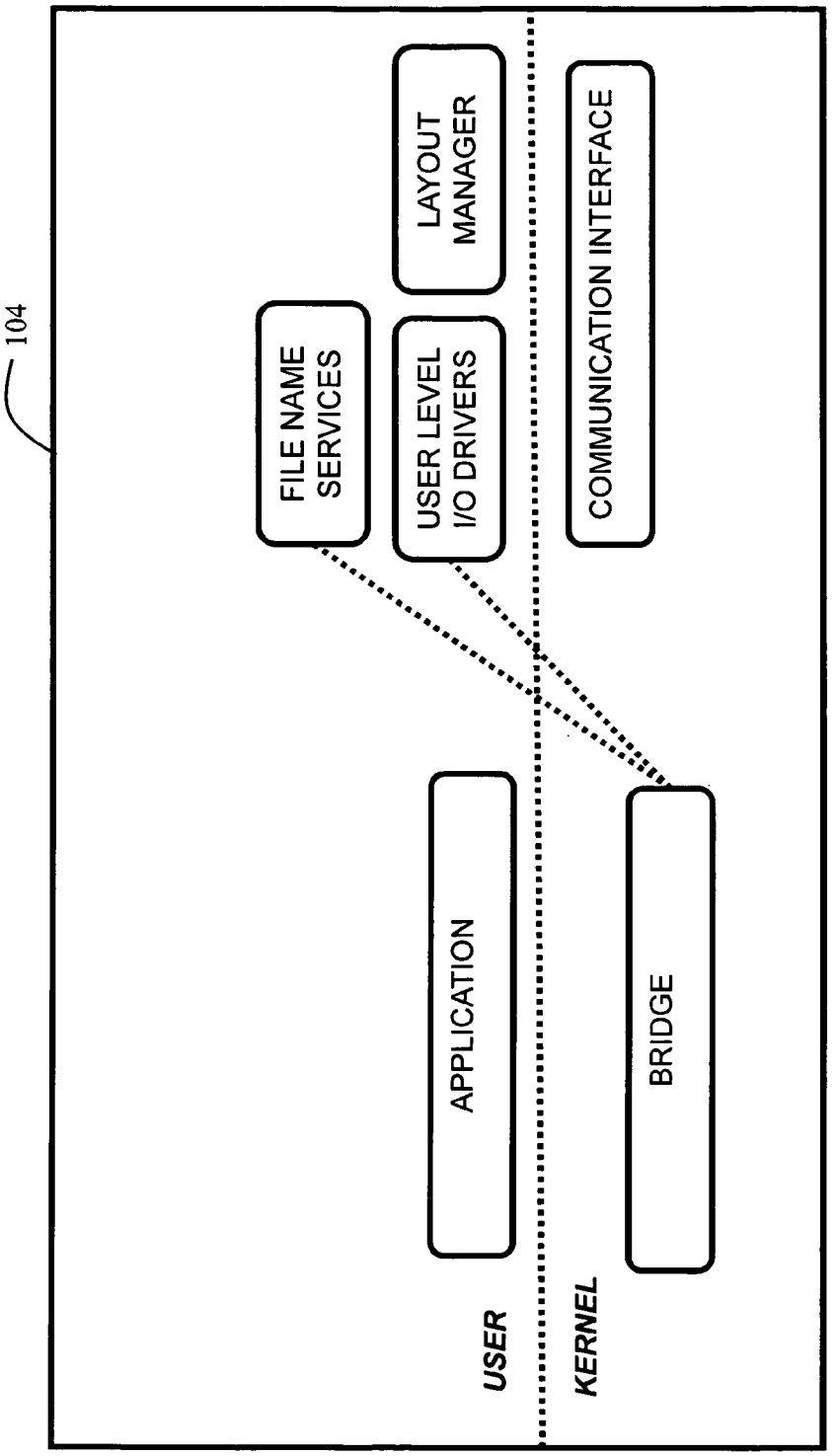

104
FILE NAME SERVICES
USER LEVEL I/O DRIVERS
LAYOUT MANAGER
COMMUNICATION INTERFACE
APPLICATION
BRIDGE
USER
KERNEL

FIGURE 21

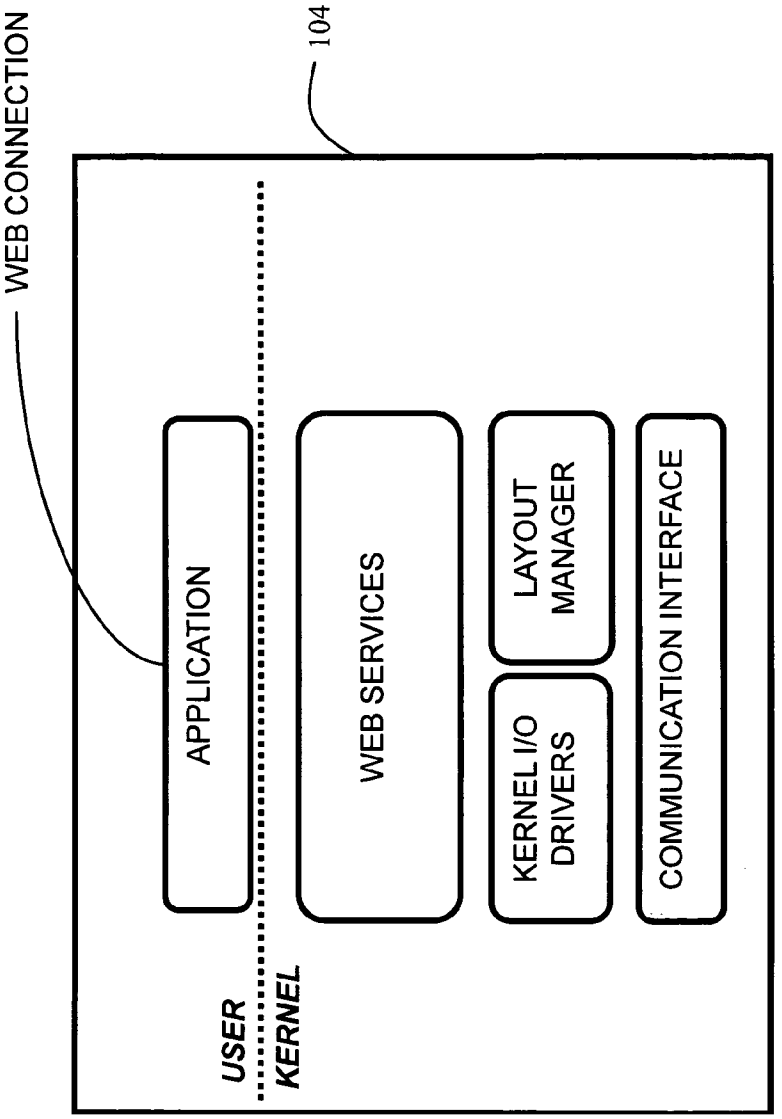

WEB CONNECTION
104
APPLICATION
USER
KERNEL
WEB SERVICES
KERNEL I/O DRIVERS
LAYOUT MANAGER
COMMUNICATION INTERFACE

FIGURE 22

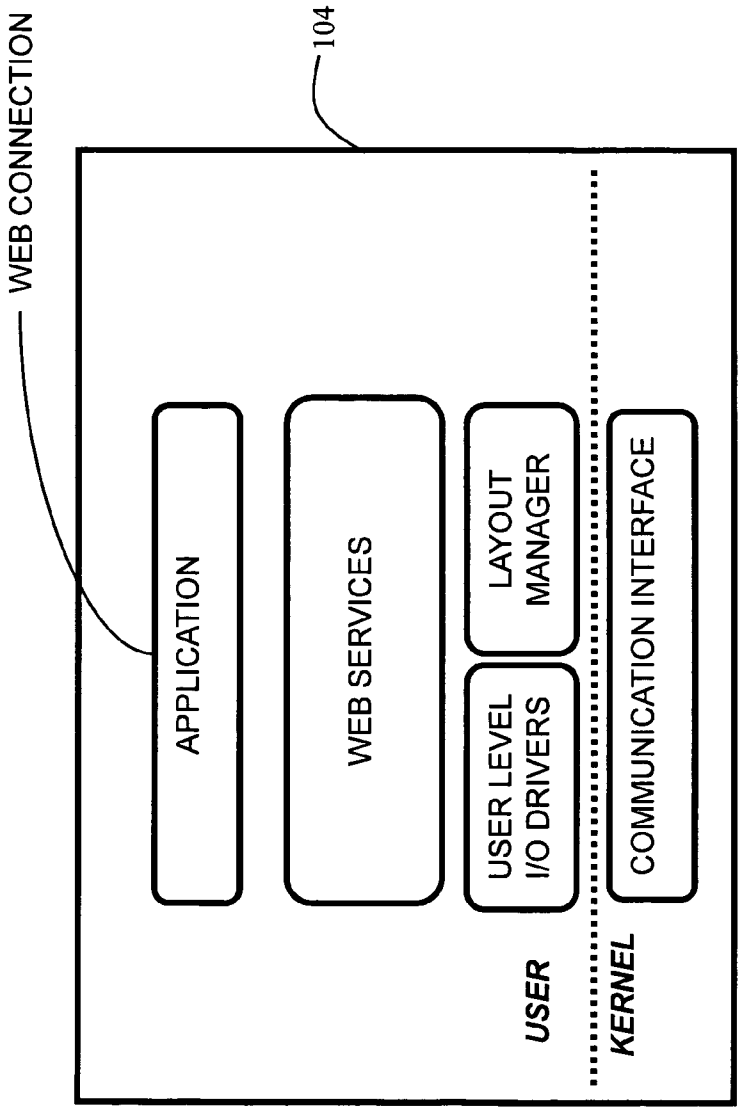

WEB CONNECTION
104
APPLICATION
WEB SERVICES
USER LEVEL I/O DRIVERS
LAYOUT MANAGER
COMMUNICATION INTERFACE
USER
KERNEL

FIGURE 23

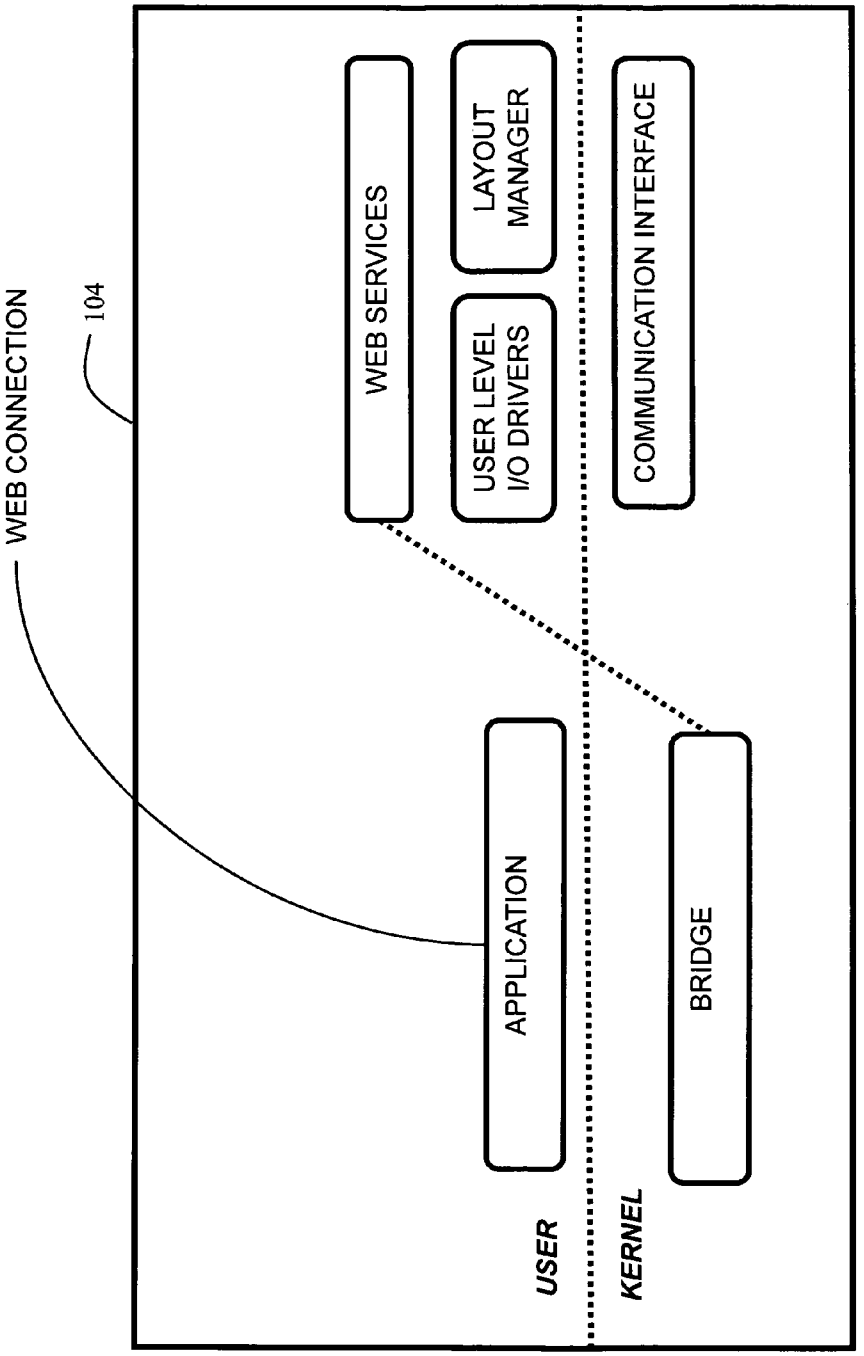

WEB CONNECTION
104
WEB SERVICES
LAYOUT MANAGER
USER LEVEL I/O DRIVERS
COMMUNICATION INTERFACE
APPLICATION
BRIDGE
USER
KERNEL

FIGURE 24

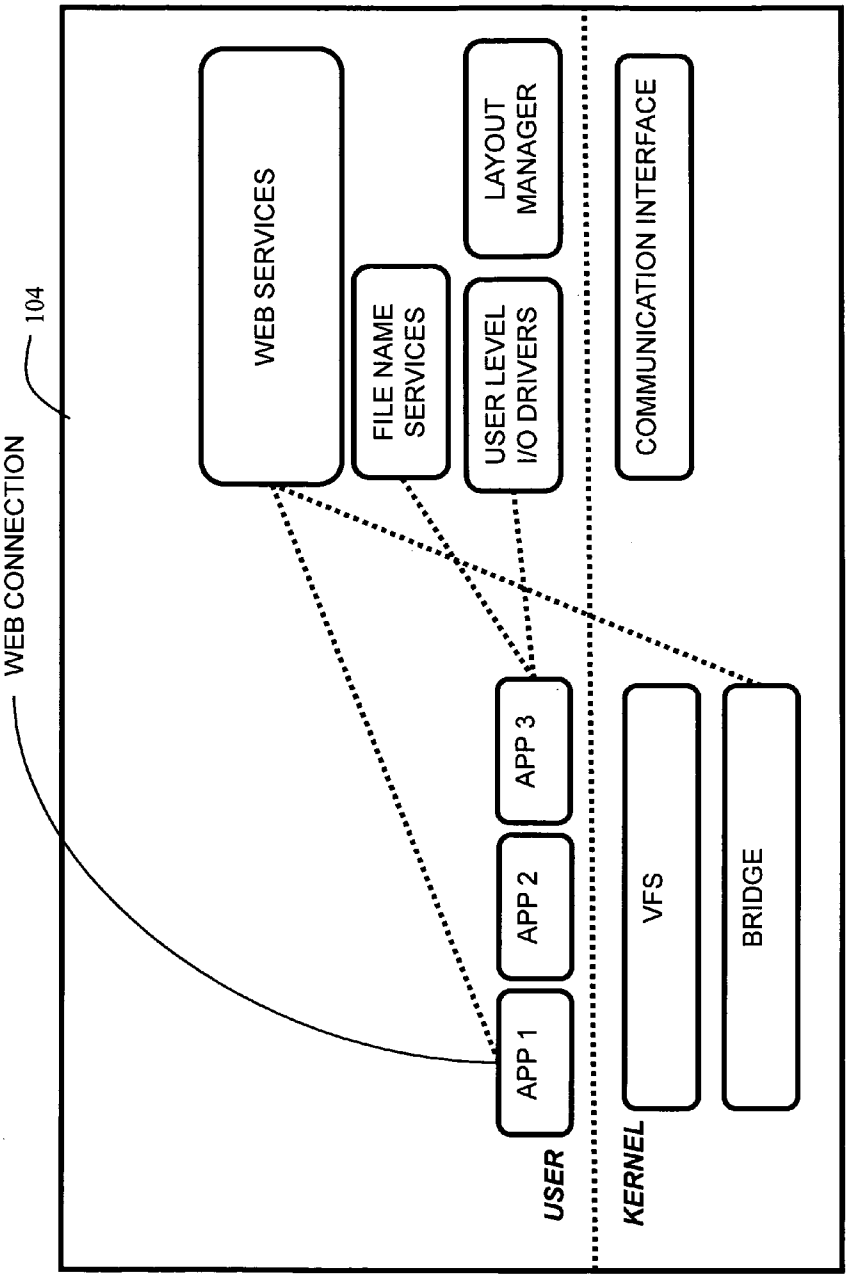

WEB CONNECTION
104
WEB SERVICES
FILE NAME SERVICES
USER LEVEL I/O DRIVERS
LAYOUT MANAGER
COMMUNICATION INTERFACE
APP 1
APP 2
APP 3
VFS
BRIDGE
USER
KERNEL

FIGURE 25

FEDERATING DATA BETWEEN GROUPS OF SERVERS

BACKGROUND OF THE INVENTION

1. Technical Field

This application relates to the field of storing data, and more particularly to the field of data storage services in a scalable high capacity system.

2. Description of Related Art

It has been estimated that the amount of digital information created, captured, and replicated in 2006 was 161 exabytes or 161 billion gigabytes, which is about three million times the information in all the books ever written. It is predicted that between 2006 and 2010, the information added annually to the digital universe will increase more than six fold from 161 exabytes to 988 exabytes. The type of information responsible for this massive growth is rich digital media and unstructured business content. There is also an ongoing conversion from analog to digital formats—film to digital image capture, analog to digital voice, and analog to digital TV.

The rich digital media and unstructured business content have unique characteristics and storage requirements that are different than structured data types (e.g. database records), for which many of today's storage systems were specially designed. Many conventional storage systems are highly optimized to deliver high performance I/O for small chunks of data. Furthermore, these systems were designed to support gigabyte and terabyte sized information stores.

In contrast, rich digital media and unstructured business content have greater capacity requirements (petabyte versus gigabyte/terabyte sized systems), less predictable growth and access patterns, large file sizes, billions and billions of objects, high throughput requirements, single writer, multiple reader access patterns, and a need for multi-platform accessibility. Conventional storage systems have met these needs in part by using specialized hardware platforms to achieve required levels of performance and reliability. Unfortunately, the use of specialized hardware results in higher customer prices and may not support volume economics as the capacity demands grow large—a differentiating characteristic of rich digital media and unstructured business content.

Some of the cost issues have been addressed with tiered storage, which attempts to reduce the capital and operational costs associated with keeping all information on a single high-cost storage tier. However, tiered storage comes with a complex set of decisions surrounding technology, data durability, functionality and even storage vendor. Tiered storage solutions may introduce unrelated platforms, technologies, and software titles having non-zero operational costs and management requirements that become strained as the quantity of data increases.

In addition, tiered storage may cause a data replica incoherence which results in multiple, disjoint copies of information existing across the tiers of storage. For example, storage management software handling data backup and recovery may make multiple copies of information sets on each storage tier (e.g. snapshots, backup sets, etc). Information Life-cycle Management (ILM) software dealing with information migration from one tier to another may create additional and often overlapping copies of the data. Replication software may make an extra copy of the information set within a particular tier in order to increase performance to accessing applications. Each of these functions typically runs autonomously from one another. The software may be unable to realize and/or take advantage of the multiple replicas of the same information set.

In addition, for large scale unstructured information stores, it may be difficult to maintain a system and manage the environment as components fail. For example, a two petabyte information store may be comprised of eight thousand 250-gigabyte disk drives. Disk failures should be handled in a different manner in a system of this scale so that the system continues to operate relatively smoothly whenever one or only a few of the disk drives fail.

The problems set forth above are addressed in published U.S. patent application no. 20090112789 titled POLICY BASED FILE MANAGEMENT, which is assigned to the assignee of the present application and is incorporated by reference herein. The system described therein provides a multi-petabyte offering for building cloud storage that combines massive scalability with automated data placement to deliver content and information services anywhere in the world. The system operates as a single entity using metadata and business policy constructs to direct content to locations and users.

In some cases, it may be desirable to federate data from two or more clouds in a way that causes the data to appear to an end user as being from a single cloud. However, this may be difficult when attempting to join public and private clouds and/or joining clouds provided by different vendors that have different structures. In addition to any technical constraints, there may be security issues that need to be addressed when a private cloud containing sensitive data is combined with a public cloud.

Thus, it would be desirable to provide a system that facilitates joining different clouds and addresses security issues associated with joining public and private clouds.

SUMMARY OF THE INVENTION

According to the system described herein, managing data includes storing metadata for the data on at least one of a plurality of servers that form a first data storage cloud and storing a first portion of the data on at least one of a plurality of servers that form a second data storage cloud that is separate from the first data storage cloud, where the data is accessed by first accessing the metadata to determine one or more locations for the data. A client that manages the data may access the first data storage cloud directly. The client may access the second data storage cloud through the first data storage cloud. The client may access the second data storage cloud directly. Managing data may also include storing a second portion of the data on at least one of a plurality of servers from the first data storage cloud, where the second portion of data is separate from the first portion of data where at least some of the second portion of data may be a mirror of the first portion of data. Managing data may also include storing a second portion of the data on at least one of a plurality of servers that form a third data storage cloud that is separate from the first data storage cloud and the second data storage cloud, where at least some of the second portion of data may be a mirror of the first portion of data.

According further to the present invention, computer software that manages data is provided in a computer-readable medium. The software includes executable code that stores metadata for the data on at least one of a plurality of servers that form a first data storage cloud and executable code that stores a first portion of the data on at least one of a plurality of servers that form a second data storage cloud that is separate from the first data storage cloud, where the data is accessed by first accessing the metadata to determine one or more locations for the data. The computer software may also include executable code that stores a second portion of the data on at

least one of a plurality of servers from the first data storage cloud, where the second portion of data is separate from the first portion of data where at least some of the second portion of data may be a mirror of the first portion of data. The computer software may also include executable code that stores a second portion of the data on at least one of a plurality of servers that form a third data storage cloud that is separate from the first data storage cloud and the second data storage cloud where at least some of the second portion of data may be a mirror of the first portion of data.

According further to the system described herein, a data storage system includes at least one client, a first plurality of interconnected servers, coupled to the at least one client, that store metadata for the data, and a second plurality of interconnected servers that store at least a first portion of the data and are coupled to the at least one client and/or the first plurality of interconnected servers, where the second plurality of interconnected servers is separate from the first plurality of interconnected servers and where the data is accessed by first accessing the metadata to determine one or more locations for the data. The at least one client may access the second plurality of interconnected servers through the first plurality of interconnected servers. The at least one client may access the second plurality of interconnected servers directly. A second portion of the data, separate from the first portion of the data, may be stored on at least one of the first plurality of interconnected servers where at least some of the second portion of data may be a mirror of the first portion of data. The data storage system may also include, a third plurality of interconnected servers that store a second portion of the data that is separate from the first portion of the data, the third plurality of interconnected servers being coupled to the at least one client, the first plurality of interconnected servers, and/or the second plurality of interconnected servers, where at least some of the second portion of data may be a mirror of the first portion of data.

According further to the system described herein, accessing data includes determining if the data is provided on a local group of servers or on an external group of servers, if the data is provided on a local group of servers, using a storage server to access the data, and, if the data is provided on an external group of servers, using a proxy server to access the data, where the proxy server interacts with an entity accessing the data in a manner that is substantially similar to interaction between the entity and the storage server. Using the proxy server may include initially providing an account id and a password. Following providing an account id and a password, using the proxy server may include using an account id and a shared secret. Using the proxy server may include using RSA ID tokens or cryptographic certificates. Metadata for the data may be provided on the local group of servers. At least some of the data may be provided on the external group of servers. The local group of servers may be a local data storage cloud. The external group of servers may be at least one external data storage cloud.

According further to the present invention, computer software that accesses data is provided in a computer-readable medium. The software includes executable code that determines if the data is provided on a local group of servers or on an external group of servers, executable code that uses a storage server to access the data if the data is provided on a local group of servers, and executable code that uses a proxy server to access the data if the data is provided on an external group of servers, where the proxy server interacts with an entity accessing the data in a manner that is substantially similar to interaction between the entity and the storage server. Executable code that uses the proxy server may initially provide an account id and a password. Following providing an account id and a password, executable code that uses the proxy server may use an account id and a shared secret. Executable code that uses the proxy server may use RSA ID tokens or cryptographic certificates. Metadata for the data may be provided on the local group of servers. At least some of the data may be provided on the external group of servers. The local group of servers may be a local data storage cloud and the external group of servers may be at least one external data storage cloud.

According further to the present invention, a data storage system includes at least one client, a local group of interconnected servers that are accessed by the client using a storage server, and an external group of servers that are accessed by the client using a proxy server, where the proxy server interacts with the client in a manner that is substantially similar to interaction between the client and the storage server. The at least one client may access the external group of servers through the local group of servers. The at least one client may access the external group of servers directly. Metadata may be provided on the local group of servers. At least some data corresponding to the metadata may be provided on the external group of servers.

BRIEF DESCRIPTION OF DRAWINGS

FIG. **1**A is a diagram illustrating servers and clients according to an embodiment of the system described herein.

FIG. **1**B is a diagram illustrating a plurality of servers according to an embodiment of the system described herein.

FIGS. **2**A, **2**B, **2**C, and **2**D are diagrams illustrating a client coupled to servers and to other network(s) according to embodiments of the system described herein.

FIG. **3** is a diagram illustrating a client having server operations software, client software, and a plurality of interfaces therebetween according to an embodiment of the system described herein.

FIG. **4** is a diagram illustrating a file having a metadata file object and a plurality of data file objects according to an embodiment of the system described herein.

FIG. **5** is a diagram illustrating a metadata file object for a file according to an embodiment of the system described herein.

FIG. **6**A is a diagram illustrating an example of a layout storage object tree for a file according to an embodiment of the system described herein.

FIG. **6**B is a diagram illustrating an example of a layout storage object tree with multiple maps for a file according to an embodiment of the system described herein.

FIG. **6**C is a diagram illustrating another example of a layout storage object tree with multiple maps and replication nodes for a file according to an embodiment of the system described herein.

FIG. **7**A is a diagram illustrating mapping to a physical storage location in a local cloud according to an embodiment of the system described herein.

FIG. **7**B is a diagram illustrating mapping to physical objects in an external cloud according to an embodiment of the system described herein.

FIG. **7**C is a diagram illustrating a client using a storage server to access a physical storage location in a local cloud according to an embodiment of the system described herein.

FIG. **7**D is a diagram illustrating a client using a proxy server to access physical objects in an external cloud according to an embodiment of the system described herein.

FIG. **8** is a flowchart illustrating obtaining data from a physical storage location indicated by a map according to an embodiment of the system described herein.

FIG. **9** is a flowchart illustrating a client obtaining a lease for and operating on a file according to an embodiment of the system described herein.

FIG. **10** is a flowchart illustrating a client reading data from a file according to an embodiment of the system described herein.

FIG. **11** is a flowchart illustrating a client writing data to a file according to an embodiment of the system described herein.

FIG. **12** is a flowchart illustrating steps performed by a client in connection with finding an alternative copy of data according to an embodiment of the system described herein.

FIG. **13** is a flowchart illustrating a client writing to synchronous mirrors for data according to an embodiment of the system described herein.

FIG. **14** is a flow chart illustrating a client converting file names to object identifiers according to an embodiment of the system described herein.

FIG. **15** is a diagram illustrating a client having an application in user memory address space and a having a VFS, file name services, kernel I/O drivers, layout manager, and a communication interface in kernel memory address space according to an embodiment of the system described herein.

FIG. **16** is a flow chart illustrating operation of a VFS at a client according to an embodiment of the system described herein.

FIG. **17** is a diagram illustrating a client having an application, file name services, user level I/O drivers, and a layout manager in user memory address space and having a communication interface in kernel memory address space according to an embodiment of the system described herein.

FIG. **18** is a diagram illustrating a client having an application, a file presentation layer, user level I/O drivers, and a layout manager in user memory address space and having a VFS and communication interface and a kernel memory address space to user memory address space bridge in kernel memory address space according to an embodiment of the system described herein.

FIG. **19** is a diagram illustrating a client having an application in user memory address space and having file name services, kernel I/O drivers, a layout manager, and a communication interface in kernel address space according to an embodiment of the system described herein.

FIG. **20** is a diagram illustrating a client having an application, file name services, user level I/O drivers, and a layout manager in user memory address space and having a communication interface in kernel memory address space according to an embodiment of the system described herein.

FIG. **21** is a diagram illustrating a client having an application, file name services, user level I/O drivers, and a layout manager in user memory address space and having a communication interface and a kernel memory address space to user memory address space bridge in kernel memory address space according to an embodiment of the system described herein.

FIG. **22** is a diagram illustrating a client having an application in user memory address space and having a Web Services module, kernel I/O drivers, a layout manager, and a communication interface in kernel memory address space according to an embodiment of the system described herein.

FIG. **23** is a diagram illustrating a client having an application, a Web Services layer, user level I/O drivers, and a layout manager in user memory address space and having a communication interface in kernel memory address space according to an embodiment of the system described herein.

FIG. **24** is a diagram illustrating a client having an application, a Web Services layer, user level I/O drivers, and a layout manager in user memory address space and having a communication interface and a kernel memory address space to user memory address space bridge in kernel memory address space according to an embodiment of the system described herein.

FIG. **25** is a diagram illustrating a client having a plurality of applications, a Web Services layer, file name services, user level I/O drivers, and a layout manager in user memory address space and having a VFS, a communication interface and a kernel memory address space to user memory address space bridge in kernel memory address space according to an embodiment of the system described herein.

DETAILED DESCRIPTION OF VARIOUS EMBODIMENTS

Referring to FIG. **1**A, a diagram illustrates servers **102** coupled to a plurality of clients **104-106**. Each of the clients **104-106** represents one or more processing devices that receives file services from the servers **102**. Each of the clients **104-106** may or may not be independent of other ones of the clients **104-106**. One or more of the clients **104-106** may be a multiprocessing/multiuser system and possibly have multiple independent users. The clients **104-106** represent any number of clients.

The file services provided by the servers **102** may include data storage and retrieval as well as related operations, such as data mirroring, cloning, etc. The servers **102** may be implemented using a plurality of services (and/or interconnected file servers including SAN components) that are provided by interconnected processing and/or storage devices. In an embodiment herein, each of the clients **104-106** may be coupled to the servers **102** using the Web, possibly in conjunction with local TCP/IP connections. However, it is possible for one or more of the clients **104-106** to be coupled to the servers **102** using any other appropriate communication mechanism and/or combinations thereof to provide the functionality described herein.

Referring to FIG. **1**B, the servers **102** are shown in more detail as including a plurality of server groups **112-114**, where each of the groups **112-114** may include one or more individual servers that may be managed together as a single data storage cloud. The terms "cloud", "data storage cloud", etc. should be generally understood herein as an integrated group of servers. Different ones of the groups **112-114** (clouds) may be managed separately from each other. As discussed in more detail elsewhere herein, the groups may be interconnected to transfer information using any appropriate means, including being interconnected through one or more of the clients **104-108**, being interconnected through the Internet, a SAN, a private LAN or WAN, directly connected, and/or using any other appropriate interconnection to provide for information transfer as discussed elsewhere herein. For the discussion herein, one of the groups **112-114** may be a local cloud that is performing operations discussed herein while another one of the groups may be an external cloud that contains data accessed by the local cloud.

Referring to FIG. **2**A, the client **104** is shown as being coupled to the servers **102** and to one or more other network(s). The other network(s) may include a local area network (LAN). Thus, the client **104** may be a gateway between the servers **102** and a LAN to which one or more other devices (not shown) may also be coupled. The client **104** may act as a

local file server to the one or more other devices coupled to the LAN by providing data from the servers **102** to the one or more other devices. Of course, it is possible for one or more other clients to simultaneous act as gateways to the same or different other network(s). Generally, for the discussion herein, reference to a particular one of the clients **104-106** may be understood to include reference to any or all of the clients **104-106** coupled to the servers **102** unless otherwise indicated.

Referring to FIG. **2**B, a diagram shows the client **104** being coupled to the servers **102** and one or more other network(s) (e.g., a LAN) in a configuration that is different from that shown in FIG. **2**A. In the configuration of FIG. **2**B, a router **118** is coupled between the servers **102** and the client **104**. The router **118** may be any conventional router that may be accessed by the client **104**. In the configuration of FIG. **2**B, the client **104** uses only a single connection point to both the servers **102** and to the other network(s). In the configuration of FIG. **2**B, the client **104** may act as local file server and gateway between the servers **102** and one or more other devices (not shown) coupled to the other network(s).

Referring to FIG. **2**C, the client **104** as shown as being used to interconnect two server groups: Group X and Group Y. The connections to Group X and/or Group Y may or may not include a router, such as the router **118** shown in FIG. **2**B and may or may not be direct or through other network configurations, as described elsewhere herein. In the embodiment of FIG. **2**C, the client **104** may communicate with either the Group X servers and/or the Group Y servers, but communication from the Group X servers to the Group Y servers is through the client **104**. One of Group X or Group Y may be a local cloud while the other is a foreign cloud.

Referring to FIG. **2**D, the client **104** as shown as being connected to two server groups: Group X and Group Y. The connections to Group X and/or Group Y may or may not include a router, such as the router **118** shown in FIG. **2**B and may or may not be direct or through other network configurations, as described elsewhere herein. In the embodiment of FIG. **2**D, the client **104** may communicate with the Group X servers and/or the Group Y servers. However, unlike the embodiment of FIG. **2**C, the Group X servers may communication with the Group Y servers without having to go through the client **104**. Just as with FIG. **2**C, one of Group X or Group Y may be a local cloud while the other is a foreign cloud.

Of course, any other appropriate connection configurations may be used by any of the client **104-106** coupled to the servers **102**, the groups **112-114**, and/or to any other network(s) and/or devices. In some embodiments, the clients **104-106** may access the metadata provided on one of the groups **112-114** and then may use the metadata to access data stored on another one of the groups **112-114**. It is also possible for one of the groups **112-114** to access data from another one of the groups **112-114** by routing data requests through one of the clients **104-106**. In such a case, the requests/data may pass through the client without any interpretation by the client.

Referring to FIG. **3**, the client **104** is shown in more detail having server operations software **122**, client software **124**, and an interface layer **125** that includes a plurality of interfaces **126-128** between the server operations software **122** and the client software **124**. The server operations software **122** facilitates the exchange of information/data between the client **104** and the servers **102** to provide the functionality described herein. In some cases, the server operations software **122** may contain proxy servers for accessing external clouds. The server operations software **122** is described in more detail elsewhere herein.

The client software **124** represents any software that may be run on the client **104**, including application software, operating system software, Web server software, etc., that is not part of the server operations software **122** or the interface layer **125**. As described in more detail elsewhere herein, it is possible to have the client software **124** interact with the servers **102** through different ones of the interfaces **126-128** at the same time.

The file services described herein may be implemented by the servers **102** using a set of file objects where a file that is accessed by the client software includes a metadata file object which points to one or more data file objects that contain the data for the file. Accessing the file would involve first accessing the metadata file object to locate the corresponding data file objects for the file. Doing this is described in more detail elsewhere herein. Note, however, that any appropriate file object mechanism may be used for the system described herein. Also, in some embodiments, a metadata file object may be provided on one of the groups of servers **112-114** (local cloud) while a corresponding one or more data file objects are provided on another one of the groups of servers **112-114** (external cloud).

Referring to FIG. **4**, a file **130** is shown as including a metadata file object **132** and a plurality of data file objects. The metadata file object **132** contains information that points to each of the data file objects **134-136**. Accessing the file includes first accessing the metadata file object **132** and then using information therein to locate the appropriate one or more of the corresponding data file objects **134-136**. As discussed elsewhere herein, in some cases, the metadata file object **132** may be provided on a different one of the groups of servers **112-114** (local cloud) than one or more of the corresponding data file objects **134-136** (external cloud).

Referring to FIG. **5**, the metadata file object **132** is shown in more detail as including an object attributes section **142** and a Layout Storage Object (LSO) tree section **144**. The object attributes section contains conventional file-type attributes such as owner id, group id, access control list, last modification time, last access time, last change time, creation time, file size, and link count. Many of the attributes are self-explanatory. The last modification time corresponds to the last time that the data for the data objects **134-136** had been modified while the last change time corresponds to when the object metadata had last been changed. The link count indicates the number of other objects that reference a particular file (e.g., aliases that point to the same file). In an embodiment herein, a file and its related objects are deleted when the link count is decremented to zero.

The LSO tree section **144** includes a data structure that includes one or more maps for mapping the logical space of the file to particular data file objects. The LSO tree section **144** may also indicate any mirrors for the data and whether the mirrors are synchronous or asynchronous. LSO trees and mirrors are described in more detail elsewhere herein.

Referring to FIG. **6**A, a simple LSO tree **160** is shown as including an LSO root node **162** and a single map **164**. The LSO root node **162** is used to identify the LSO tree **160** and includes links to one or more map(s) used in connection with the file corresponding to the LSO tree **160**. The map **164** maps logical locations within the file to actual data storage location. A process that accesses logical storage space of a file represented by the LSO tree **160** first uses the LSO root node **162** to find the map **164** and then uses the map **164** to translate logical addresses within the file to an actual data storage locations. As discussed in more detail elsewhere herein, the map **164** may point to physical storage space in the same one of the server groups **112-114** that contains the physical stor-

age space for the LSO tree **160**. Alternatively, the map **164** may point to objects in storage space in a different one of the server groups **112-114** than the one of the server groups **112-114** that contains the physical storage space for the LSO tree **160**.

Referring to FIG. **6**B, an LSO tree **170** is shown as including an LSO root node **172** and a plurality of maps **174-176**. Each of the maps **174-176** may represent a different range of logical offsets within the file corresponding to the LSO tree **170**. For example, the map **174** may correspond to a first range of logical offsets in the file. The map **174** may map logical locations in the first range to a first actual storage device. The map **175** may correspond to a second range of logical offsets in the file, different than the first range, which may be mapped to a different actual storage device or may be mapped to the same actual storage device as the map **174**. Similarly, the map **176** may correspond to a third range of logical offsets in the file, different than the first range and the second range, which may be mapped to a different actual storage device or may be mapped to the same actual storage device as the map **174** and/or the map **175**. Note that some of the maps **174-176** may or may not point to physical storage space in the same one of the server groups **112-114** that contains the physical storage space for the LSO tree **170** while other ones of the maps **174-176** may or may not point to objects in physical storage space in a different one of the server groups **112-114** than the one of the server groups **112-114** that contains the physical storage space for the LSO tree **170**.

Referring to FIG. **6**C, an LSO tree **180** is shown as including an LSO root node **181** and a pair of replication nodes **182***a*, **182***b*, which indicate that the underlying data is to be mirrored (replicated) and which indicate whether the mirror is synchronous or asynchronous. Synchronous and asynchronous mirrors are discussed in more detail elsewhere herein. The node **182***a* has a plurality of children maps **183-185** associated therewith while the node **182***b* has a plurality of children maps **186-188** associated therewith. The replication nodes **182***a*, **182***b* indicate that the data corresponding to the maps **183-185** is a mirror of data corresponding to the maps **186-188**. In some embodiments, the nodes **182***a*, **182***b* may be implemented using a single node **189** to indicate replication.

A process accessing a file having the LSO tree **180** would traverse the tree **180** and determine that data is mirrored. As discussed in more detail elsewhere herein, depending upon the type of mirroring, the process accessing the LSO tree **180** would either write the data to the children of both of the nodes **182***a*, **182***b* or would provide a message to another process/server (e.g., the servers **102**) that would perform the asynchronous mirroring. Mirroring is discussed in more detail elsewhere herein.

Note that, just as with the maps **164**, **174-176**, discussed above, some of the maps **183-189** may or may not point to physical storage space in the same one of the server groups **112-114** that contains the physical storage space for the LSO tree **180** while other ones of the maps **183-189** may or may not point to objects in physical storage space in a different one of the server groups **112-114** than the one of the server groups **112-114** that contains the physical storage space for the LSO tree **180**. Note also, however, that it may be advantageous in some instances to have the maps **183-185** for the replication node **182***a* point to objects on one of the server groups **112-114** while the maps **186-189** for the other replication node **182***b* point to physical objects on another one of the server groups **112-114**.

In some embodiments, it may be beneficial to provide physical storage for all LSO trees on a first one of the server groups **112-114** (e.g. a local cloud) while providing physical storage for some or all of the corresponding data on a second, different, one of the server groups **112-114** (e.g., an external cloud). The first one of the server groups **112-114** may be a private cloud accessed by a particular organization while the second one of the server groups **112-114** is a public cloud that is accessed by many organizations, such as the Amazon S3 public cloud. Alternatively, the first one of the server groups **112-114** may be a public cloud while the second one of the server groups **112-114** is a private cloud or both the first and the second one of the server groups **112-114** could be public clouds or could be private clouds. The LSO trees may be provided on an external cloud. In addition, the data may be provided on separate clouds so that a first portion is provided on one cloud and a second (or subsequent) portion is provided on a second (or subsequent) cloud where each of the clouds that contain data are separate from each other.

As described herein, the federation of a plurality of clouds allows the data to appear to a user (client) as if the data were provided on a single cloud. Note that since the LSO trees provide meaningful structure to the data, then maintaining the LSO trees in a private cloud provides some security even though some or all of the corresponding data may be provided in a public cloud. Note also that the physical storage space required for the LSO trees is expected to be much less than that required for the corresponding data. Accordingly, in instances where the LSO trees are provided in a private cloud while the corresponding data is provided in a public cloud, the physical storage space that needs to be maintained for the private cloud is much less than it would be otherwise while sensitive metadata may be maintained securely in the private cloud.

Referring to FIG. **7**A, the map **164** (described above in connection with FIG. **6**A) is shown as pointing to a physical storage **192** that is provided on a local cloud. The map **164** may represent any of the other ones of the maps described herein and/or may represent any appropriate mapping mechanism for accessing physical storage on a local cloud. For example, the map **164** may contain an identifier for the physical storage **164** in addition to some type of offset and/or additional identifier to indicate a particular portion of the physical storage **192**. There may also be a length (or similar) value indicating an amount of data that corresponds to the map **164**. As discussed elsewhere herein, security for local cloud access may be handled by another mechanism, and thus it is not necessary for the map **164** to contain security information, although in some embodiments it may be useful to have security information be included with the map **164**.

Referring to FIG. **7**B, the map **164** is shown as pointing to physical storage (objects) in one or more external clouds. In such a case, the map **164** may contain or point to information used to access the objects in the external cloud **194**, which of course depends upon the particular access mechanism employed by the external cloud. For example, in some systems an account id and a password could be used. There may also be additional information, such as file/object identifier(s), subaccount information, etc. In an embodiment herein, once a connection to data in the external cloud **194** has been established, subsequent communications with the external cloud **194** may include at least some of the information (e.g., an account id) along with a shared secret. Other possible authentication/security techniques may be used, including RSA ID tokens, cryptographic certificates, etc.

In an embodiment herein, the map **164**, as well as any other maps that are used, point to a single object provided on the external cloud **194** which corresponds to a single file in the file system of the external cloud **194**. In other embodiments, it is possible to provide multiple objects in a single file in the

file system of the external cloud **194**. It is even possible to provide objects from different sources (e.g., different users, accounts, private clouds, etc.) into a single file. However, in that case, it may be necessary to handle any security issues that are created by this.

Referring to FIG. **7**C, the client **104** is shown using a storage server **196** to access the physical storage **192** containing data from the local cloud, as discussed elsewhere herein. The storage server **196** provides data to the client **104** and may represent any combination of software and hardware (including at least a portion of the server operations software **122** that is part of the client **104**, discussed above). The client **104** may represent any client or other device/mechanism that accesses the servers **102** to exchange data therewith. The storage server **196** may provide a specific interface to the client **104** and to software used by the client **104**.

Referring to FIG. **7**D, the client **104** is shown using a proxy server **198** to access the external clouds **194**. The proxy server **198** provides data to the client **104** and may represent any combination of software and hardware (including at least a portion of the server operations software **122** that is part of the client **104**, discussed above). The proxy server **198** may interact with the client **104** and to software used by the client **104** in a manner that is substantially similar (and possibly identical) to the interaction between the client **104** and the storage server **196**. The proxy server **198** may exchange information with the external cloud **194** using the REST (Representational State Transfer) protocol, which is known.

In some embodiments, it may be possible to have data provided in a local cloud and for that data to point to additional data in an external cloud.

In an embodiment herein, the map **164** includes a flag (or similar) to indicate whether the data pointed to by the map **164** is provided on a local cloud or an external cloud. In instances where the data is provided on a local cloud, the storage server **196** (or similar) is used. In instances where the flag indicates that the data is provided in an external cloud, the proxy server **198** is used. Once one of the servers **196**, **198** is selected, operation of the client **104** and related components is identical, or nearly so. Accordingly, the system provided herein may provide a federation of clouds that is transparent to a client accessing the servers **102**.

Referring to FIG. **8**, a flow chart **200** illustrates in detail steps performed in connection with obtaining data from a physical storage location indicated by the map **164**. Processing begins at a first step where it is determined if the data is part of a local cloud or stored in an external cloud, as discussed elsewhere herein. If the data is not external, then control transfers from the test step **201** to a step **202** where the storage server **196** is used to access the data according to the information provided in the map **164**. Following the step **201**, processing is complete.

If it is determined at the test step **201** that the data is located in an external cloud, then control transfers from the step **201** to a step **203** where an element used to iterate through the available proxy servers is set to point to the first one of the proxy servers. In an embodiment herein, each of the proxy servers may be provided with different capabilities so that, for example, one set of proxy servers can access external cloud X but not external cloud Y, another set of proxy servers can access external cloud Y but not external cloud X, yet another set can access both external clouds, etc. Furthermore, different proxy servers may have different capabilities such as speed, efficiency, cost, etc. that could make one proxy server more desirable than another in certain situations. Accordingly, there may be certain criteria imposed that render only some of the proxy servers suitable for accessing the external data. For example, is the external data is located on cloud X, then only proxy servers capable of accessing cloud X are suitable and satisfy the criteria. Note also that it is possible for a user/administrator to indicate that certain proxy servers are suitable/acceptable for certain types of accesses.

Following the step **203** is a test step **204** where it is determined if the proxy server pointed to by the element used to iterate through proxy servers satisfies whatever criteria that is imposed. If so, then control passes from the test step **204** to a test step **205** where the proxy server is used to access the data. As discussed elsewhere herein, the proxy server may provide an account id and password and/or an account id and shared secret in connection with accessing the data. In an embodiment herein, the external cloud does not rely on any security characteristics imposed by the cloud/client from which the request is generated. Thus, for example, an administrative user for one cloud may still need to provide the same security information as any other user when accessing an external cloud. Note also that the security information needed to access the external cloud may be stored with the map **164**, pointed to by the map **164**, or stored in some other location. Following the step **205**, processing is complete.

If it is determined at the test step **204** that the proxy server indicated by the iteration pointer does not satisfy the criteria, then control transfers from the test step **204** to a step **206** where the iteration pointer is incremented to point to the next proxy server. Following the step **206** is a test step **207** where it is determined if the pointer points past the end of the proxy servers (i.e., all of the available proxy servers have been examined to determine if any of them meet the specified criteria). If all of the proxy servers have not been examined, then control passes from the test step **207** back to the step **204** for another iteration. Otherwise, control passes from the step **207** to a step **208** where error processing is performed. The error processing performed at the step **208** can be any appropriate processing, including returning an indicator that the data is not available. Note that there could be many reason why the data is not available, including the criteria being too restrictive (e.g., requiring a transfer speed that is not available), one or more of the proxy servers being off-line, etc. Following the step **208**, processing is complete.

For the system described herein, file objects are accessed by one of the clients **104-106** by first requesting, and obtaining, a lease from the servers **102**. The lease corresponds to the file objects for the particular file being accessed and to the type of access. A lease may be for reading, writing, and/or for some other operation (e.g., changing file attributes). In an embodiment herein, for objects corresponding to any particular file, the servers **102** may issue only one write lease at a time to any of the clients **104-106** but may issue multiple read leases simultaneously and may issue read lease(s) at the same time as issuing a write lease. However, in some embodiments it may be possible to obtain a lease for a specified logical range of a file for operations only on that range. Thus, for example, it may be possible for a first client to obtain lease for writing to a first logical range of a file while a second client may, independently, obtain a lease for writing to a second and separate logical range of the same file. The two write leases for different logical ranges may overlap in time without violating the general rule that the system never issues overlapping write leases for the same data.

The lease provided to the clients **104-106** from the servers **102** includes security information (security token) that allows the client appropriate access to the data. The security token may expire after a certain amount of time. In an embodiment herein, a client accesses data by providing an appropriate security token for the data as well as client users/ownership

information. Thus, for example, a user wishing to access data would first obtain a lease and then would provide the access request to the servers **102** along with the security token and information identifying the owner (client) accessing the data. The servers **102** would then determine whether the access requested by the client was permissible. After the lease expires (the security token expires), the user requests the lease again. Data security may be implemented using conventional data security mechanisms.

After obtaining a lease for accessing a file, a client may then cache the corresponding metadata, including the LSO tree, into local storage of the client. The client may then use and manipulate the local cached version of the metadata and may use the metadata to obtain access to the data. As described in more detail elsewhere herein, a client does not directly modify metadata stored by the servers **102** but, instead, sends update messages to the servers **102** to signal that metadata for a file may need to be modified by the servers **102**.

Referring to FIG. **9**, a flowchart **210** illustrates steps performed by a client in connection with requesting a lease for a file (objects associated with a file) for performing operations thereon. Processing begins at a first step **212** where the client requests the lease for the file. As discussed in more detail elsewhere herein, a client requesting a lease includes specifying the type of access (e.g., read, write, etc.). Following the step **212** is a test step **214** where it is determined if the request has been granted. If not, then control transfers from the test step **214** to a step **216** where processing is performed in connection with the lease not being granted to the client. The particular processing performed at the step **216** may include, for example, providing an error message to the client process requesting access to the file corresponding to the lease and/or waiting for an amount of time and then retrying the request. Note that it is possible that a lease for a particular file is not available at one time is subsequently available at another time because, for example, the lease is released by another client in between the first request and the second request. In any event, any appropriate processing may be performed at the step **216**. Following the step **216**, processing is complete.

If it is determined at the test step **214** that the lease requested at the step **212** has been granted, then control transfers from the test step **214** to a step **218** where the client performs an operation using the file for which the lease was granted. Operations performed at the step **218** include reading data and/or writing data. Different types of processing that may be performed at the step **218** are described in more detail elsewhere herein.

Following the step **218** is a test step **222** where it is determined if the operations performed at the step **218** require an update. In some instances, a client may obtain a lease and perform operations that do not affect the file or the underlying file objects. For example, a client may acquire a lease for reading a file and the operation performed at the step **218** may include the client reading the file. In such a case, no update may be necessary since the file and corresponding file objects (metadata, data objects, etc.) have not changed. On the other hand, if the client obtains a lease for writing data the file and the operation performed at the step **218** includes writing data to the file, then the underlying file objects will have been changed and an update message needs to be sent the servers **102**. If it is determined at the test step **222** that an update is necessary, then control passes from the test step **222** to a step **224** where an update message is sent by the client to the servers **102**.

Following the step **224**, or following the step **222** if no update is necessary, control passes to a test step **226** where it is determined if the client is finished with the file. In some instances, the client may perform a small number of operations on the file, after which the client would be finished with the file at the step **226**. In other cases, the client may be performing a series of operations and may not yet have completed all of the operations.

If it is determined at the test step **226** that the client is not finished with the file, then control passes from the test step **226** to a test step **228** where it is determined if the lease for the file has expired. Note that a lease may be provided by the servers **102** to the client with a particular expiration time and/or the associated security token may expire. In addition, it may be possible for the servers **102** to recall leases provided to clients under certain circumstances. In either case, the lease may no longer be valid. Accordingly, if it is determined at the step **228** that the lease has expired (and/or has been recalled by the servers **102**), then control passes from the test step **228** back to the step **212** request the lease again. Otherwise, if the lease has not expired, then control passes from the test step **228** back to the step **218** to perform another iteration.

If it is determined at the test step **226** that the client is finished with the file, then control passes from the test step **226** to a step **232** where the client releases the lease by sending a message to the servers **102** indicating that the client no longer needs the lease. Once the client releases the lease, it may be available for other clients. Following the step **232**, processing is complete.

In an embodiment herein, data file objects may be indicated as having one of four possible states: current, stale, immutable, or empty. The current state indicates that the data object is up to date and current. The stale state indicates that the data is not valid but, instead, requires updating, perhaps by some other process. In some instances, the stale state may be used only in connection with mirror copies of data (explained in more detail elsewhere herein). Data may be stale because it is a mirror of other data that was recently written but not yet copied. The immutable state indicates that the corresponding data is write protected, perhaps in connection with a previous clone (snapshot) operation. The empty state indicates that no actual storage space has yet been allocated for the data.

Referring to FIG. **10**, a flow chart **240** illustrates steps performed by a client in connection with performing read operations after obtaining a read lease for a file. Processing begins at a first test step **242** where it is determined if the data object being read is in the current state. If not, then control transfers from the test step **242** to a step **244** where it is determined if the data object being read is in the immutable state. If it is determined at the step **244** that the data object being read is in the immutable state or if it is determined at the test step **242** that the data object being read is in the current state, then control transfers to a step **246** where the read operation is performed.

A client may read file data by providing the appropriate data file object identifier to the servers **102** as well as providing appropriate security credentials. Accordingly, the read operation performed at the step **246** includes the client sending an appropriate request to the servers **102** and waiting for a result therefrom. Alternatively, if the file data is stored in a different one of the groups **112-114** than the processor performing the processing illustrated by the flow chart **240**, then performing a read operation at the step **246** may include providing a client ID, account info, and credentials to the different one of the groups **112-114**.

Following the step **246** is a test step **248** where it is determined if the servers **102** have returned a result indicating that the data file object is unavailable. In some cases, a data file object that is otherwise current or immutable may neverthe-

less become unavailable. For example, the physical storage space that holds the data file object may become temporarily disconnected and/or temporarily busy doing some other operation or, if a data file object may be stored on a different one of the groups **112-114** that is unavailable. If it is determined at the test step **248** that the data file object is available, then control transfers from the test step **248** to a test step **252** where it is determined if the read operation was successful. If so, then control transfers from the test step **252** to a step **254** where the result of the read operation is returned to the process at the client that caused the read operation to be performed. The result may include the data that was read and a status indicator. Following the step **254**, processing is complete.

If it is determined at the test step **252** that the read operation performed at the step **246** was not successful, then control transfers from the test step **252** to a step **256** where error processing is performed. The particular error processing performed at the step **256** is implementation dependent and may include, for example, reporting the error to a calling process and/or possibly retrying the read operation a specified number of times. Following the step **256**, processing is complete.

If it is determined at the test step **244** that the data object being read is not in the immutable state, then control transfers from the test step **244** to a test step **258** where it is determined if the data object is in the stale state. If not, then, by virtue of the test steps **242**, **244**, **258** and process of elimination, the data object is in the empty state. In an embodiment herein, reading a data object in the empty state causes zeros to be returned to the calling process. Accordingly, if it is determined at the test step **258** that the data object is not in the stale state, then control transfers from the test step **258** to a step **262** where zeros are returned in response to the read operation. Following the step **262**, processing is complete.

If it is determined at the test step **258** that the data file object is in the stale state, or if it is determined at the test step **248** that the data file object is not available, then control transfers to a test step **264** to determine if an alternative version of the data file object is available for reading. As discussed in more detail elsewhere herein, there may be multiple versions of the same data file objects that exist at the same time due to mirroring. Accordingly, if the data file object being read is in the stale state or otherwise unavailable, it may be possible to read a mirror copy of the data file object that may be in the current state. The test performed at the step **264** is described in more detail elsewhere herein.

If it is determined at the test step **264** that an alternative version of the data file object is available, then control transfers from the test step **264** to a step **266** where the alternative version of the data file object is selected for use. Following the step **266**, control transfers back to the test step **242** for another iteration with the alternative data file object.

If it is determined at the test step **264** that an alternative version of the data file object is not available, then control transfers from the test step **264** to a step **268** where the client process waits. In an embodiment herein, it may be desirable to wait for a data file object to become current and/or available. Following the step **268**, control transfers back to the step **242** for another iteration. Note that, instead of waiting at the step **268**, processing may proceed from the step **264** to the step **256** to perform error processing if there is no alternative data file object available. In other embodiments, it may be possible to perform the step **268** a certain number of times and then, if the data file object is still unavailable or in the stale state and there is no alternative data file object, then perform the error processing at the step **256**.

Referring to FIG. **11**, a flow chart **280** illustrates steps performed by a client in connection with performing write operations after obtaining a write lease for a file. Processing begins at a first test step **282** where it is determined if the data file object to which the write is being performed is in the immutable state. If so, then control transfers from the step **282** to a step **284** where new actual storage space is allocated for the data file object to avoid overwriting the immutable data. Allocating new storage space for a data object may include providing an appropriate request to the servers **102**. In instances where the file data is stored in a different one of the groups **112-114** than the processor performing the processing illustrated by the flow chart **280**, then allocating new storage space at the step **284** may include providing a client ID, account info, and credentials to the different one of the groups **112-114**. Following the step **284**, control transfers back to the step **282** to begin the processing for the write operation again.

If it is determined at the step **282** that the data file object to which the write is being performed is not in the immutable state, then control transfers from the step **282** to a step **286** where it is determined if the data file object to which the write is being performed is in the stale state. If not, then control transfers from the test step **286** to a test step **288** where it is determined if the data file object to which the write is being performed is in the empty state. If so, then control transfers from the step **288** to the step **284**, discussed above, where new physical storage space is allocated. Following the step **284**, control transfers back to the step **282** to begin the processing for the write operation again.

If it is determined at the step **288** that the data file object to which the write is being performed is not in the empty state, then control transfers from the test step **288** to a step **292** where the write operation is performed. Note that the step **292** is reached if the data file object to which the write operation is being performed is not in the immutable state, not in the stale state, and not in the empty state (and thus is in the current state). A client writes file data by providing the appropriate data file object location identifier to the servers **102** as well as providing appropriate security credentials. Accordingly, the write operation performed at the step **292** includes the client sending an appropriate request to the servers **102** and waiting for a result therefrom. As with the read operation, discussed above, if the file data is stored in a different one of the groups **112-114** than the processor performing the processing illustrated by the flow chart **280**, then performing a write operation at the step **292** may include providing a client ID, account info, and credentials to the different one of the groups **112-114**.

Following the step **292** is a test step **294** where it is determined if the write operation performed at the step **292** was successful. If so, then control transfers from the test step **294** to a test step **296** where it is determined if there are synchronous mirrors of the data file object to which the write is being performed. The test performed at the step **296** may include, for example, determining if a parent node of the data file object in the file LSO tree indicates replication. If not, then control transfers from the test step **296** to a step **298** where an update (message) is sent to the servers **102** indicating that the write had been performed. Following the step **298**, processing is complete.

If it is determined at the test step **296** that there are synchronous mirrors of the data file object to which the write is being performed, then control passes from the test step **296** to a step **302** where the data that was written at the step **292** is also written to the synchronous mirror(s). The processing performed at the step **302** is discussed in more detail elsewhere herein. Following the step **302**, control transfers to the

step **298**, discussed above, where an update (message) is sent to the servers **102**. Following the step **298**, processing is complete.

If it is determined at the test step **294** that the write operation performed at the step **292** was not successful, or if it is determined at the test step **286** that the data file object to which the write operation is being performed is in the stale state, then control transfers to a step **304** where the data file object to which the write is attempting to be performed is removed from the client's local copy of the LSO tree. At the end of the write operation illustrated by the flow chart **280**, the client may inform the servers **102** (at the step **298**) of the difficulty in writing to the data object so that the servers **102** can take appropriate action, if necessary.

Following the step **304** is a test step **306** where it is determined if an alternative version of the data is available. As discussed in more detail elsewhere herein, there may be multiple versions of the same data file objects that exist at the same time due to mirroring. Accordingly, if the data file object to which the write operation is being performed is stale or otherwise cannot be written to, it may be possible to write to a mirror copy of the data. The test performed at the step **306** is like the test performed at the step **264** and is described in more detail elsewhere herein. If it is determined at the test step **306** that an alternative version of the data corresponding to the data file object is available, then control transfers from the test step **306** to a step **308** where the alternative version is selected for writing. Following the step **308**, control transfers back to the test step **282** for another iteration with the alternative data file object.

If it is determined at the test step **306** that an alternative version of the data corresponding to the data file object is not available, then control transfers from the test step **306** to a step **312** to perform error processing if there is no alternative available. The particular error processing performed at the step **312** is implementation dependent and may include, for example, reporting the error to a calling process and/or possibly retrying the write operation a specified number of times before reporting the error. Following the step **312**, control transfers to the step **298**, discussed above, to send update information to the servers **102**. Following the step **298**, processing is complete.

Referring to FIG. **12**, a flow chart **320** illustrates in more detail steps performed in connection with the alternative available test step **264** of FIG. **10** and/or the alternative available test step **306** of FIG. **11**. Processing begins at a first test step **322** where it is determined if the file has any mirror data file objects at all. In some instances, a file may not use mirrors, in which case there would be no alternative copy available. Accordingly, if it is determined at the test step **322** that the file does not have any mirror data file objects, then control transfers from the test step **322** to a step **324** where a value is returned indicating that no alternative copies are available. Following the step **324**, processing is complete.

If it is determined at the test step **322** that mirror copies are available, then control transfers from the test step **322** to a step **326** where a pointer is made to point to metadata for a first mirror data file object. For the processing discussed herein, a pointer may be used to iterate through metadata for mirror data file objects to find a useable data file object. Following the step **326** is a test step **328** where it is determined if the pointer is past the end of the list of mirror data file objects (has iterated through all of the metadata for mirror data file objects). If so, then control passes from the test step **328** to the step **324**, discussed above, to return a value that indicates that no alternatives are available.

If it is determined at the test step **328** that the pointer is not past the end of a list of mirror data file objects, then control transfers from the test step **328** to a test step **332** where it is determined if the pointer points to metadata indicating that the corresponding data file object in a stale state. If so, then control transfers from the test step **332** to a step **334** where the pointer is made to point to metadata for the next data file object to be examined. Following the step **334**, control transfers back to the step **328**, discussed above, for another iteration. If it is determined at the test step **332** that the pointer does not point to metadata indicating that the corresponding data file object in the stale state, then control transfers from the test step **332** to a step **336** where the metadata indicating the data file object that is pointed to by the pointer is returned as an alternative data file object that may be used by the calling process. Following the step **336**, processing is complete.

Referring to FIG. **13**, a flow chart **350** illustrates in more detail operations performed in connection with the step **302** of the flow chart **280** of FIG. **11** where data that has been written is copied to a number of synchronous mirrors (mirror data file objects). Processing begins at a first step **352** where a pointer that is used to iterate through metadata for the mirror data file objects is set to point to metadata for the first one of the mirror data file objects. Following the step **352** is a test step **354** where it is determined if the pointer used for iterating through the metadata for the mirror data file objects points past the end (i.e., if all of the mirror data file objects have been processed). If so, then processing is complete. Otherwise, control transfers from the test step **354** to a test step **356** where it is determined if the status indicated by metadata for the corresponding mirror data file object pointed to by the pointer indicates that the mirror data file object is current. If not, then control passes from the test step **356** to a test step **358** where it is determined if the status indicated by the metadata for the mirror data file object pointed to by the pointer indicates that the mirror data file object is in the stale state. If so, then control passes from the test step **358** to a step **362** where metadata for the mirror data file object is removed from the client's local copy of the LSO tree. In an embodiment herein, a synchronous mirror data file object should not be in a stale state and, if that occurs, it may indicate an error condition. Accordingly, following the step **362** is a step **364** where information about the stale mirror is sent to the servers **102**, which may perform recovery processing in response thereto.

Note that if a mirror data file object is neither in the stale state nor in the current state, then the mirror data file object is either in the empty state or in the immutable state. In either case, it may be necessary to allocate new space for a data file object to which the data is to be written. Accordingly, if it is determined at the test step **358** that metadata indicates that the corresponding data file object is not in the stale state, then control passes from the test step **358** to a step **366** where new space is allocated for the mirror data file object (similar to the step **284** of FIG. **11**, discussed above). Following the step **366** is a step **368** where the data that is being copied across synchronous mirror data file objects is written to the mirror data file object pointed to by the pointer used to iterate through the metadata for the mirror data file objects (similar to the step **292** of FIG. **11**, discussed above). Note that the step **368** may also be reached from the test step **356** if it is determined that the mirror data file object is current. Following the step **368** is a step **372** where the pointer used to iterate through metadata for the mirror data file objects is made to point to the next one. Note that the step **372** is also reached following the step **364**. Following the step **372**, control transfers back to the test step **354** for another iteration.

The system described herein may access file objects using object identifiers. In an embodiment herein, each file object that is stored among the servers **102**, including file objects from both local and external clouds, may be assigned a unique object identifier that identifies each file object and distinguishes each file object from other file objects in the system. However, many applications use a file naming structure and/or a hierarchical directory to access files and data therein. For example, a file name "C:\ABC\DEF\GHI.doc" indicates a file called "GHI.doc" stored in a sub-directory "DEF" that is stored in another directory "ABC" located on a root volume "C". A nested directory structure may be provided by implementing directories as special files that are stored in other directories. In the example given above, the sub-directory "DEF" may be implemented as a file stored in the directory "ABC".

The system described herein may present to applications a conventional naming structure and directory hierarchy by translating conventional file names into file object identifiers. Such a translation service may be used by other services in connection with file operations. In an embodiment herein, each directory may include a table that correlates file names and sub-directory names with file object identifiers. The system may examine one directory at a time and traverse sub-directories until a target file is reached.

Referring to FIG. **14**, a flow chart **380** illustrates steps performed in connection with providing a file name translation service (file name service) that translates a conventional hierarchical file name into a file object identifier. The file name service may receive a conventional hierarchical file name as an input and may return an object identifier (or, in some cases, an error). Processing begins at a first step **382** where the file name service receives a file name, such as a conventional hierarchical file name. Following the step **382** is a test step **384** where it is determined if the syntax of the file name is OK. Mechanisms for checking the syntax of a hierarchical file name are know in the art and include, for example, checking that only appropriate characters have been used. If it is determined at the test step **384** that the syntax is not OK, then control transfers from the test step **384** to a step **386** where an error indicator (error message) is returned to the calling process. Following the step **386**, processing is complete.

If it is determined at the test step **384** that the syntax of the provided name is OK, then control transfers from the test step **384** to a step **388** where the root directory is read. In an embodiment herein, all file name paths begin at a single common root directory used for all file objects stored in the servers **102**. In other embodiments, there may be multiple root directories where specification of a particular root directory may be provided by any appropriate means, such as using a volume identifier, specifically selecting a particular root directory, etc.

Following the step **388** is a test step **392** where it is determined if the target file (or sub-directory that is part of the file name path) is in the directory that has been read. If not, then control passes from the test step **392** to the step **386**, discussed above, where an error is returned. In some embodiments, the file-not-found error that results from the test at the step **392** may be different from the syntax error that results from the test at the step **384**.

If it is determined that the target file or a sub-directory that is part of the file name path is in the directory that has just been read, then control passes from the test step **392** to a test step **394** where it is determined if the directory that has just been read contains the target file (as opposed to containing a sub-directory that is part of the file name path). If so, then control passes from the test step **394** to a step **396** where the object identifier of the target file object is returned to the calling process. Following the step **396**, processing is complete.

If it is determined at the test step **394** that the directory that has just been read contains a sub-directory that is part of the file name path, then control transfers from the test step **394** to a step **398** where the sub-directory is read so that the sub-directory becomes the directory being examined. In effect, processing at the step **398** traverses the chain of subdirectories to eventually get to the target file. Following the step **398**, control transfers back to the step **392**, discussed above, for a next iteration.

Referring to FIG. **15**, a diagram shows the client **104** as including user address memory space and kernel address memory space. In an embodiment herein, user address memory space is memory space that is generally used by user applications and related processes while kernel address memory space is memory space that is generally accessible only by system processes, such as an operating system kernel and related processes. As discussed in more detail herein, it is possible to have different portions of the system described herein reside and operate in the user memory space and/or the kernel memory space. In addition, it is possible for the client **104** to have multiple different interfaces to access file objects at the servers **102**.

In FIG. **15**, the client **104** is shown as including an application in the user memory address space and a virtual file system (VFS), file name services, kernel I/O drivers, a layout manager, and a communication interface in the kernel memory address space. The VFS is an abstraction layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. The VFS allows the application running on the client **104** to access file objects on the servers **102** without the application needing to understand the details of the underlying file system. The VFS may be implemented in a conventional fashion by translating file system calls by the application into file object manipulations and vice versa. For example, the VFS may translate file system calls such as open, read, write, close, etc. into file object calls such as create object, delete object, etc.

The VFS may use the file name services, described elsewhere herein, to translate file names into object identifiers. The kernel I/O drivers provide an interface to low-level object level I/O operations. The kernel I/O drivers may be modeled after, and be similar to, Linux I/O drivers. The layout manager may perform some of the processing on LSO trees corresponding to files, as discussed in more detail elsewhere herein. The communication interface provides communication between the client **104** and the servers **102**.

The communication interface may be implemented using any appropriate communication mechanism. For example, if the client **104** communicates with the servers **102** via an Internet connection, then the communication interface may use TCP/IP to facilitate communication between the servers **102** and the client **104**. In instances where objects from one of the groups **112-114** may be accessed by a client from another one of the groups **112-114**, the communication interface may include an appropriate mechanism to formulate data accesses to a different group. For example, the communication interface may include a mechanism for providing a client ID, account info, and credentials to the different one of the groups **112-114**.

The application of FIG. **15** may correspond to the client software **124** of FIG. **3**. The VFS of FIG. **15** may correspond to one of the interfaces **126-128** of FIG. **3**. The file name services, kernel I/O drivers, layout manager, and communi-

cation interface of FIG. **15** may correspond to the server operations software **122** of FIG. **3**. Similar correlation between components of FIG. **3** and other figures may also be found.

Referring to FIG. **16**, a flow chart **410** illustrates steps performed by a VFS to provide file services in connection with an application running on the client **104**. Processing begins at a first step **412** where a file system operation requested by an application may be translated into one or more object operations. For example, a file operation to open a file for reading may be converted to object operations that include obtaining an object lease for reading as discussed elsewhere herein. Following the step **412** is a step **414** where the VFS translates the file name into an object identifiers using the file name services discussed above in connection with FIG. **14**. Operations that follow may be performed using the object identifiers obtained at the step **414**.

Following the step **414** is a test step **416** where it is determined if the requested operation requires the LSO tree. As discussed elsewhere herein, operations such as read, write, etc. use LSO trees corresponding to file objects. However, some possible file operations may not require accessing a corresponding LSO tree. If it is determined at the test step **416** that the LSO tree is needed, then control transfers from the test step **416** to a step **418** where the VFS accesses the LSO manager to perform the necessary operations. For example, for a read operation, the LSO manager may perform processing like that illustrated in the flow chart **240** of FIG. **10**. Following the step **418**, or following the step **416** if the LSO is not needed, is a step **422** where the operations are passed to low level kernel I/O drivers (e.g., via one or more appropriate API's). The kernel I/O drivers use the communication module to communicate between the client **104** and the servers **102** in connection with performing the requested operation(s). In instances where the application running on the client **104** has requested data and/or other information from the servers **102** (in the same or a different one of the groups **112-114**), the data and/or information may be passed back up through the communication interface, kernel I/O drivers, etc. to the VFS and ultimately to the application. As discussed elsewhere herein, the communication module may use and/or include one or more storage servers and/or one or more proxy servers.

Referring to FIG. **17**, the client **104** is shown as having an application, file name services, user level I/O drivers, and a layout manager all provided in user memory address space. The functionality of the VFS that was shown in FIG. **15** and described above may be performed instead by library routines linked to the application, and thus are part of the application. These routines would provide functionality like that discussed above in connection with FIG. **16**. Accordingly, it is the application that uses the file name services and makes calls to the user level I/O drivers (like the kernel I/O drivers) and to the layout manager. The communication interface is still maintained in the kernel memory address space.

Note that, for the configuration of FIG. **15**, modifications are provided by modifying system processes (the operating system), which is disadvantageous for a number of reasons. For example, if the client **104** is a multiuser computing system, then modifying the operating system may involve restarting the entire system and thus disrupting all of the users. In contrast, the configuration of FIG. **17** is advantageous since it allows modification of the system in the application/user memory address space so that the operating system of the client **104** does not need to be modified. However, the configuration of FIG. **17** does not use a VFS, and thus does not obtain the advantageous separation of the application from the file system that is provided by the VFS in FIG. **15**.

Referring to FIG. **18**, the client **104** is shown as having an application in user memory address space that accesses file objects through a VFS in kernel memory address space like that illustrated in FIG. **15**. However, the file name services, I/O drivers, and the layout manager all reside in the user memory address space like the system illustrated in FIG. **17**. The VFS communicates with components in the user memory address space through a bridge between kernel memory address space and user memory address space, such as a FUSE (or similar) interface. The bridge allows file system components to be provided in user memory space instead of kernel address memory space while still preserving the VFS in the kernel address memory space. Thus, the configuration illustrated by FIG. **18** provides the advantages of using a VFS, as illustrated in the configuration of FIG. **15**, along with the advantages of having file system components in the user address memory space, as illustrated in the configuration of FIG. **17**.

It is possible in some instances to have applications and/or other processing in the user memory address space of the client **104** access file objects directly, rather than through a file services layer like the VFS and/or equivalent functionality provided by user linkable libraries (e.g., the configuration illustrated in FIG. **17**). Accessing file objects directly may include invoking routines that create objects, read objects, modify objects, delete objects, etc. Accessing file objects may also include, if the objects are stored in a different one of the groups **112-114**, providing a client ID, account info, and credentials to the different one of the groups **112-114**. If an application and/or other process in the user memory address space of the client **104** accesses file objects directly, the application would need to know how to interpret and/or manipulate the object data, which may not always be desirable. For example, an application that accesses file objects through the VFS may not need to take into account (or even know about) the structure of an LSO tree while an application that accesses objects directly may need to use the LSO tree. On the other hand, removing the file services layer may provide an opportunity for optimizations not otherwise available. Note that, since the servers **102** exchange object information/operations with the clients **104-106**, the servers **102** may not need to distinguish or be able to distinguish between application on the clients **104-106** using a file system interface (file services like the VFS) and those that are not.

Referring to FIG. **19**, the client **104** is shown as including an application in the user memory address space and kernel I/O drivers, a layout manager, and file name services in the kernel memory address space. The configuration illustrated in FIG. **19** is like that illustrated in FIG. **15**, except that the VFS is not used. In the configuration illustrated in FIG. **19**, the application could directly access the file name services, the kernel I/O drivers, and the layout manager. The communication interface in the kernel memory address space communicates with the servers **102** just as in other configurations. The direct access illustrated in FIG. **19** allows applications to manipulate file objects (via, for example, appropriate API's) while access via the VFS (or similar) allows applications to accesses file objects indirectly through file system calls to the VFS.

Referring to FIG. **20**, the client **104** is shown as having an application, user level I/O drivers, a layout manager, and file name services all provided in user memory address space. The configuration shown in FIG. **20** is like that shown in FIG. **17**. However, as set forth above, the configuration of FIG. **17** includes file service libraries that are linked into, and thus part of, the application. In contrast, in the configuration of FIG. **20**, the application is not linked into libraries with extensive

file services. Instead, like the application of the configuration illustrated in FIG. **19**, the application in the configuration of FIG. **20** uses minimal file services and, instead, uses and operates upon file objects directly using the user level I/O drivers, the layout manager and, if a file name translation is needed, the file name services.

Referring to FIG. **21**, the client **104** is shown as having an application in user memory address space and a bridge in the kernel memory address space. File name services, user level I/O drivers, and a layout manager are provided in user memory address space. However, unlike the configuration of FIG. **20**, the application does not make direct calls to the file system components in the user memory address space. Instead, the application calls the file system components indirectly through the bridge. Just as with the configuration illustrated in FIG. **18**, the configuration of FIG. **21** advantageously locates file system components in the user memory address space and, at the same time, provides a kernel memory address space layer between the application and the file system components.

Referring to FIG. **22**, the client **104** is shown as having an application in user memory address space and a Web Services module in kernel memory address space. The application may be a Web server application or any application that handles communication with the Web. In an embodiment herein, the application allows communication with the client **104**, which acts as a Web server to other computing devices (not shown) that access the client **104** through a Web connection.

The configuration illustrated in FIG. **22** provides Web Services in a manner similar to the file services and/or file object access provided by other configurations. However, the Web Services receives requests/data via a Web data protocol, such as HTML, and provides responses/data also in a Web data protocol, which may be the same or different from the protocol used for requests/data. Operations handled by the Web Services may include object-level operations such as create object, delete object, read object, modify object, modify object metadata, etc. It is also possible to provide more file system level operations, via the Web Services, that open files, read data from files, etc. by including at least some of the functionality of the file services, described elsewhere herein, with the Web Services. The Web Services may present to the other computing devices a conventional well-known Web Services protocol, such as REST or SOAP, or may provide any other appropriate protocol.

Referring to FIG. **23**, the client **104** is shown as having an application, Web Services, user level I/O drivers, and a layout manager in user memory address space. The application may include a Web connection that allows communication with the client **104**, which acts as a Web server to other computing devices (not shown) that access the client **104** through the Web connection. The configuration of FIG. **23** is like that of FIG. **17** and FIG. **20**. The advantages of the configuration shown in FIG. **23** over the configuration shown in FIG. **22** is that, generally, changes to the configuration shown in FIG. **23** do not require reconfiguring kernel memory address space processes.

Referring to FIG. **24**, the **104** is shown as having an application, Web Services, user level I/O drivers, and a layout manager in user memory address space. The application may include a Web connection that allows communication with the client **104**, which acts as a Web server to other computing devices (not shown) that access the client **104** through the Web connection. A bridge is provided in the kernel memory address space. The configuration of FIG. **24** has similar advantages to the configuration shown in FIG. **23**, but also has the advantages provided by providing the bridge, discussed elsewhere herein.

Referring to FIG. **25**, the client **104** is shown as having a plurality of applications in user memory address space, each of which may use a different interface to access file objects of the servers **102**. Each of the applications shown in FIG. **25** is meant to represent one or more applications. Accordingly, APP**1** may present one or more applications that access file objects at the servers **102** using a Web Services interface. The APP**1** application may include a Web connection that allows communication with the client **104**, which acts as a Web server to other computing devices (not shown) that access the client **104** through the Web connection. APP**2** may represent one or more applications that access file objects at the servers **102** using the VFS, and APP**3** may represent one or more applications that directly operate on file objects at the servers **102**. The different interfaces may operate at the client **104** at the same time.

Note that may other combinations of configurations, including illustrated configurations, are possible so that the client **104** may simultaneously present to applications thereon different interfaces. For example, it is possible to combine the configurations illustrated in FIGS. **15**, **19**, and **22** and/or combine the configurations of FIGS. **17**, **20**, and **23**. Other combinations, including combinations of only two illustrated configurations, are also possible. The servers **102** provide the file objects to the clients **104** provided: 1) the requesting client has appropriate authorization for whatever operation is requested for the file objects; and 2) there is no conflict with any previous request. For example, in systems where only one client is allowed to write to an object at any one time, the servers **102** would not allow one of the clients **104-106** to modify a particular object while another one of the clients **104-106** is also modifying the object.

The system described herein may be used with any server, or any group of servers, capable of providing file objects to clients. The particular form of the file objects may vary without departing from the spirit and scope of the invention. In some instances, the order of steps in the flow charts may be modified, where appropriate. The system described herein may be implemented using a computer program product/software provided in a computer-readable storage medium.

While the invention has been disclosed in connection with various embodiments, modifications thereon will be readily apparent to those skilled in the art. Accordingly, the spirit and scope of the invention is set forth in the following claims.

What is claimed is:

1. A method of accessing data, comprising:

determining if the data is provided on a local group of servers or on an external group of servers;

if the data is provided on a local group of servers, using a storage server to access the data; and

if the data is provided on an external group of servers that are not directly accessible by an entity, using a proxy server to access the data using security information provided on at least one of the local group of servers, wherein the proxy server interacts with the entity accessing the data in a manner that is substantially similar to interaction between the entity and the storage server, wherein the data on the external group of servers is not provided on the local group of servers and wherein metadata for the data on the external group of servers is provided on the local group of servers and wherein the local group of servers is accessed without using the proxy server.

2. A method, according to claim 1, wherein using the proxy server includes initially providing the security information as an account id and a password.

3. A method, according to claim 2, wherein, following providing an account id and a password, using the proxy server includes using an account id and a shared secret.

4. A method, according to claim 1, wherein using the proxy server includes using one of: RSA ID tokens and cryptographic certificates.

5. A method, according to claim 1, wherein the local group of servers is a local data storage cloud.

6. A method, according to claim 5, wherein the external group of servers is at least one external data storage cloud.

7. Computer software, provided in a non-transitory computer-readable medium, that accesses data, the software comprising:

executable code that determines if the data is provided on a local group of servers or on an external group of servers;

executable code that uses a storage server to access the data if the data is provided on a local group of servers; and

executable code that uses a proxy server to access the data if the data is provided on an external group of servers that are not directly accessible by an entity, the proxy server using security information provided on at least one of the local group of servers, wherein the proxy server interacts with the entity accessing the data in a manner that is substantially similar to interaction between the entity and the storage server, wherein the data on the external group of servers is not provided on the local group of servers and wherein metadata for the data on the external group of servers is provided on the local group of servers and wherein the local group of servers is accessed without using the proxy server.

8. Computer software, according to claim 7, wherein executable code that uses the proxy server initially provides the security information as an account id and a password.

9. Computer software, according to claim 8, wherein, following providing an account id and a password, executable code that uses the proxy server uses an account id and a shared secret.

10. Computer software, according to claim 7, wherein executable code that uses the proxy server uses one of: RSA ID tokens and cryptographic certificates.

11. Computer software, according to claim 7, wherein the local group of servers is a local data storage cloud and wherein the external group of servers is at least one external data storage cloud.

12. A data storage system, comprising:

at least one client;

a local group of interconnected servers that are accessed by the client using a storage server; and

an external group of servers, not directly accessible by the client, that are accessed by the client using a proxy server that uses security information provided on at least one of the local group of servers, wherein the proxy server interacts with the client in a manner that is substantially similar to interaction between the client and the storage server and wherein the data on the external group of servers is not provided on the local group of servers and metadata for the data on the external group of servers is provided on the local group of servers and wherein the local group of servers is accessed by the client without using the proxy server.

13. A data storage system, according to claim 12, wherein the at least one client accesses the external group of servers through the local group of servers.

14. A data storage system, according to claim 12, wherein the at least one client accesses the external group of servers directly.

15. A method, according to claim 1, wherein metadata for data provided on the external group of servers is provided on the local group of servers.

16. Computer software, according to claim 7, wherein metadata for data provided on the external group of servers is provided on the local group of servers.

17. A data storage system, according to claim 12, wherein metadata for data provided on the external group of servers is provided on the local group of servers.

* * * * *