



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

(21) International Application Number:
PCT/US2017/032004

(22) International Filing Date:
10 May 2017 (10.05.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/333,978 10 May 2016 (10.05.2016) US
62/353,320 22 June 2016 (22.06.2016) US
62/353,322 22 June 2016 (22.06.2016) US

(71) Applicant: NASUNI CORPORATION [US/US]; 313 Speen Street, Natick, MA 01760 (US).

(72) Inventors: SHAW, David, M.; c/o Nasuni Corporation, 313 Speen Street, Natick, MA 01760 (US). MCDONALD, Matthew, M.; c/o Nasuni Street, 313 Speen Street, Natick, MA 01760 (US). NEUFELD, Russell, A.; c/o Nasuni Corporation, 313 Speen Street, Natick, MA 01760 (US). LACASSE, Christopher, S.; c/o Nasuni Corporation, 313 Speen Street, Natick, MA 01760 (US).

(74) Agent: JUDSON, David, H.; Law Office Of David H. Judson, 15950 Dallas Parkway, Suite 225, Dallas, TX 75248 (US).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM,

(54) Title: VERSIONING USING EVENT REFERENCE NUMBER IN A CLOUD-BASED DATA STORE AND LOCAL FILE SYSTEMS

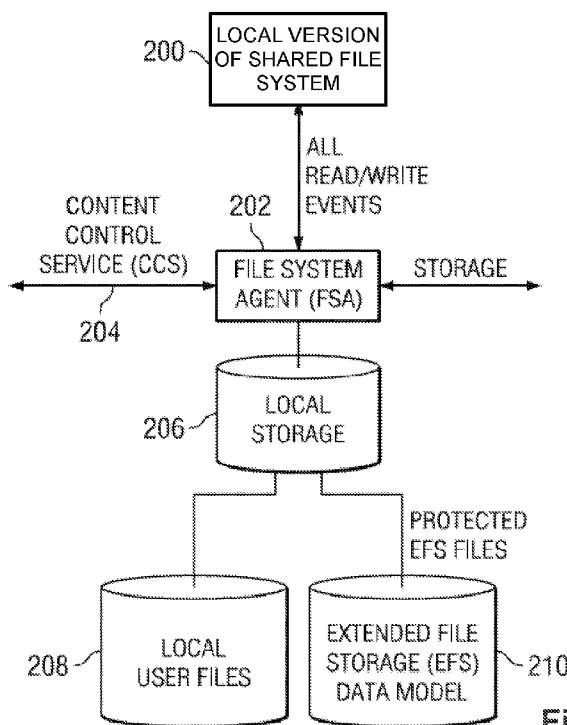


Fig. 2

(57) Abstract: A cloud-based write-once object store is configured to store inode-based data exported to the store from an enterprise file system. For each version of data (e.g., a file) exported to the store, there is a version of the inode corresponding to that data. As versions of the data are exported to the cloud, the system creates multiple versions of the inode. The set of inode versions corresponding to the versions of the file have a pointer associated therewith that specifies the latest version of the data associated with the inode. The inode versions in the set share the same pointer. The inode versions represent a revision history for the inode. For each inode version corresponding to a version of the data, information is received and stored in a new portion of the object store. The inode version for a file version comprises a list of data chunks for the file.

WO 2017/197012 A1

PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

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

Declarations under Rule 4.17:

- *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
- *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

- *with international search report (Art. 21(3))*
- *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

VERSIONING USING EVENT REFERENCE NUMBER IN A CLOUD-BASED DATA STORE AND LOCAL FILE SYSTEMS

Technical Field

[0001] The present application relates generally to data storage, and more particularly to synchronizing, updating and maintaining a versioned data store in a cloud based network-attached file system.

Background

[0002] It is known to provide an interface between an existing local file system and a data store (e.g., a “write-once” store) to provide a “versioned” file system. The versioned file system comprises a set of structured data representations, such as XML. In a representative embodiment, at a first time, the interface creates and exports to a data store a first structured data representation corresponding to a first version of the local file system. The first structured data representation is an XML tree having a root element, a single directory (the “root directory”) under the root element, zero or more directory elements associated with the root directory, and zero or more elements (such as files) associated with a given directory element. Each directory in turn can contain zero or more directories and zero or more files. Upon a change within the file system (e.g., file creation, file deletion, file modification, directory creation, directory deletion and directory modification), the interface creates and exports a second structured data representation corresponding to a second version of the file system. The second structured data representation differs from the first structured data representation up to and including the root element of the second structured data representation. Thus, the second structured data representation differs from the first structured data representation in one or more (but not necessarily all) parent elements with respect to the structured data element in which the change within the file system occurred. The interface continues to generate and export structured data representations to the data store, preferably at given “snapshot” times when changes within the file system have occurred. The data store comprises any type of back-end storage device, system or architecture. In one embodiment, the data store comprises one or more cloud storage service providers. As necessary, a given structured data representation is then used to retrieve an associated version of the file system. In this manner, the versioned file system only requires write-once behavior from the data store to preserve its complete state at any point-in-time.

[0003] A problem with the above system is that a change to any file or directory in the file system causes a new version of each parent directory all the way up to the root. This causes additional processing time and resources to create each new “version” of the file system. Also, to determine what file or directory has changed between versions of the file system, the entire directory structure needs to be “walked.” In a large file system with a large user base, the processing overhead required to maintain this directory structure is significant. It would be desirable to create versions of a more granular portion of a file system without having to create a snapshot of the entire file system.

Summary

[0004] A problem with the above system is that a change to any file or directory in the file system causes a new version of each parent directory all the way up to the root. This causes additional processing time and resources to create each new “version” of the file system. Also, to determine what file or directory has changed between versions of the file system, the entire directory structure needs to be “walked.” In a large file system with a large user base, the processing overhead required to maintain this directory structure is significant. It would be desirable to create versions of a more granular portion of a file system without having to create a snapshot of the entire file system.

In the Drawings

[0005] For a more complete understanding of the disclosed subject matter and the advantages thereof, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

[0006] Figure 1 is a block diagram illustrating how a shared versioned file system interfaces a local version of the shared versioned file system to an object-based data store;

[0007] Figure is a block diagram of a representation implementation of a portion of the interface shown in Figure 1;

[0008] Figure 3 is a more detailed implementation of the interface where there are a number of local versions of the shared versioned file system of different types;

[0009] Figure 4 illustrates a filer server implemented as an appliance within a local processing environment;

[0010] Figure 5 is a block diagram of the architecture of a shared versioned file system according to an embodiment;

[0011] Figure 6 illustrates the portion of the tree (as shown in Figure 5) after a change to the contents of a file has occurred in the local version of the shared versioned file system;

[0012] Figure 7 is a block diagram of a system for running a shared versioned file system according to an embodiment;

[0013] Figure 8 is a flow chart of a method for sending updated data to cloud storage according to an embodiment;

[0014] Figure 9 illustrates a table of updates to a shared versioned file system maintained by the filer server;

[0015] Figure 10 is a flow chart of a method for synchronizing updates from a local version of a shared versioned file system to a cloud data store according to an embodiment;

[0016] Figure 11 is a flow chart of a method for synchronizing updates from a cloud-based a shared versioned file system to a local version of same according to an embodiment

[0017] Figure 12 is a simplified illustration of a representative shard according to an embodiment;

[0018] Figure 13 is a simplified illustration of the representative shard from Figure 12 after an update according to an embodiment;

[0019] Figure 14 is a simplified illustration of a representative directory entry in the representative shard of Figure 13; and

[0020] Figure 15 depicts a representative cloud layout for an inode-based write once object store according to this disclosure.

Detailed Description

[0021] Figure 1 illustrates a simplified system 10 for providing a shared versioned file system. The system 10 includes local versions 100, 101 of the shared versioned file system and an object-based data store 120. Although not meant to be limiting, the object-based store 120 can be a “write-once” store and may comprise a “cloud” of one or more storage service providers.

Each interface or filer server 110, 111 exposes a respective local version 100, 101 of a “shared versioned file system” that only requires write-once behavior from the object-based data store 120 to preserve substantially its “complete” state at any point-in-time. As used herein, the phrase “point-in-time” should be broadly construed, and it typically refers to periodic “snapshots” of the local version of the shared versioned file system or periodic snapshots of any updates to the local version of the shared versioned file system (e.g., once every “n” minutes). The value of “n” and the time unit may be varied as desired. Each filer server 100, 101 provides for a local version 100, 101 of the shared versioned file system that has complete data integrity to the cloud. In particular, this solution circumvents the problem of a lack of reliable atomic object replacement in cloud-based object repositories. The filer servers 100, 101 are not limited for use with a particular type of back-end data store. When the filer servers 100, 101 are positioned in “front” of data store 120, the filer servers 100, 101 have the effect of turning whatever is behind it into respective local versions of a “shared versioned file system” (“SVFS”). The SVFS is a construct that is distinct from the filer server itself, and the SVFS continues to exist irrespective of the state or status of the filer server (from which it may have been generated). Moreover, the SVFS is self-describing, and it can be accessed and managed separately from the back-end data store, or as a component of that data store. Thus, the SVFS (comprising a set of structured data representations) is location-independent. In one embodiment, the SVFS resides within a single storage service provider (SSP) although, as noted above, this is not a limitation. In another embodiment, a first portion of the SVFS resides in a first SSP, which a second portion resides in a second SSP. Generalizing, any given SVFS portion may reside in any given data store (regardless of type), and multiple VFS portions may reside across multiple data store(s). The SVFS may reside in an “internal” storage cloud (i.e. a storage system internal to an enterprise), an external storage cloud, or some combination thereof.

[0022] The interface or filer server 104 can be implemented as a machine. A representative implementation is the NASUNI® Filer, available from Nasuni Corporation of Massachusetts. Thus, for example, typically the filer server 104 is a rack-mounted server appliance comprising of hardware and software. The hardware typically includes one or more processors that execute software in the form of program instructions that are otherwise stored in computer memory to comprise a “special purpose” machine for carrying out the functionality described herein. Alternatively, the filer server 104 is implemented as a virtual machine or

appliance (e.g., via VMware®, or the like), as software executing on a server, or as software executing on the native hardware resources of the local version of the SVFS. The filer server 104 serves to transform the data representing the local version of the SVFS (a physical construct) into another form, namely, a shared versioned file system comprising a series of structured data representations that are useful to reconstruct the shared versioned file system to any point-in-time.

[0023] Although not meant to be limiting, preferably each structured data representation is an XML document (or document fragment). As is well-known, extensible markup language (XML) facilitates the exchange of information in a tree structure. An XML document typically contains a single root element (or a root element that points to one or more other root elements). Each element has a name, a set of attributes, and a value consisting of character data, and a set of child elements. The interpretation of the information conveyed in an element is derived by evaluating its name, attributes, value, and position in the document.

[0024] The filer server 104 generates and exports to the write-once data store a series of structured data representations (e.g., XML documents) and data objects that together comprise the shared versioned file system. The structured data representations are stored in the data store 120. Preferably, the XML representations are encrypted before export to the data store. The transport may be performed using known techniques. In particular, REST (Representational State Transfer) is a protocol commonly used for exchanging structured data and type information on the Web. Another such protocol is Simple Object Access Protocol (SOAP). Using REST, SOAP, or some combination thereof, XML-based messages are exchanged over a computer network, normally using HTTP (Hypertext Transfer Protocol) or the like. Transport layer security mechanisms, such as HTTP over TLS (Transport Layer Security), may be used to secure messages between two adjacent nodes. An XML document and/or a given element or object therein is addressable via a Uniform Resource Identifier (URI). Familiarity with these technologies and standards is presumed.

[0025] Figure 2 is a block diagram of a representative implementation of how the interface or filer server 110/111 captures all (or given) read/write events from a local version of shared versioned file system 200. In this example implementation, the interface comprises a file system agent (FSA) 202 that is positioned within a data path between a local version of shared

versioned file system 200 and its local storage 206. The file system agent 202 has the capability of “seeing” all (or some configurable set of) read/write events output from the local file system. The interface/filer server also comprises a content control service (CCS) 204 as will be described in more detail below. The content control service is used to control the behavior of the file system agent. The object-based data store is represented by the arrows directed to “storage” which, as noted above, typically comprises any back-end data store including, without limitation, one or more storage service providers. The local version of the shared versioned file system stores local user files (the data) in their native form in cache 208. Reference numeral 210 represents that portion of the cache that stores pieces of metadata (the structured data representations, as will be described) that are exported to the back-end data store (e.g., the cloud).

[0026] Figure 3 is a block diagram illustrating how the interface may be used with different types of local file system architectures. In particular, Figure 3 shows the CCS (in this drawing a Web-based portal) controlling three (3) FSA instances. Once again, these examples are merely representative and they should not be taken to limit the invention. In this example, the file system agent 306 is used with three (3) different local versions of the shared versioned file system: NTFS 300 executing on a Windows operating system platform 308, MacFS (also referred to as “HFS+” (HFSPlus)) 302 executing on an OS X operating system platform 310, and EXT3 or XFS 304 executing on a Linux operating system platform 312. These local versions of the shared versioned file system may be exported (e.g., via CIFS, AFP, NFS or the like) to create a NAS system based on VFS. Conventional hardware, or a virtual machine approach, may be used in these implementations, although this is not a limitation. As indicated in Figure 3, each platform may be controlled from a single CCS instance 314, and one or more external storage service providers may be used as an external object repository 316. As noted above, there is no requirement that multiple SSPs be used, or that the data store be provided using an SSP.

[0027] Figure 4 illustrates the interface/filer server implemented as an appliance within a local processing environment. In this embodiment, the version of 400 for the local version of the shared versioned file system is received over Ethernet and represented by the arrow identified as “NAS traffic.” That traffic is provided to smbd layer 402, which is a SAMBA file server daemon that provides CIFS (Window-based) file sharing services to clients. The layer 402 is managed by the operating system kernel 404 in the usual manner. In this embodiment, the local version of the

shared versioned file system is represented (in this example) by the FUSE kernel module 406 (which is part of the Linux kernel distribution). Components 400, 402 and 404 are not required to be part of the appliance. The file transfer agent 408 of the interface is associated with the FUSE module 406 as shown to intercept the read/write events as described above. The CCS (as described above) is implemented by a pair of modules (which may be a single module), namely, a cache manager 410, and a volume manager 412. Although not shown in detail, preferably there is one file transfer agent instance 408 for each volume of the local file system. The cache manager 410 is responsible for management of “chunks” with respect to a local disk cache 414. This enables the interface or filer server described herein to maintain a local cache of the data structures (the structured data representations) that comprise the shared versioned file system. The volume manager 412 maps the root of the FSA data to the cloud (as will be described below), and it further understands the one or more policies of the cloud storage service providers. The volume manager also provides the application programming interface (API) to these one or more providers and communicates the structured data representations (that comprise the shared versioned file system) through a transport mechanism 416 such as cURL. cURL is a library and command line tool for transferring files with URL syntax that supports various protocols such as FTP, FTPS, HTTP, HTTPS, SCP, SFTP, TFTP, TELNET, DICT, LDAP, LDAPS and FILE. cURL also supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, cookies, user + password authentication, file transfer resume, proxy tunneling, and the like. The structured data representations preferably are encrypted and compressed prior to transport by the transformation module 418. The module 418 may provide one or more other data transformation services, such as duplicate elimination. The encryption, compression, duplicate elimination and the like, or any one of such functions, are optional. A messaging layer 420 (e.g., local socket-based IPC) may be used to pass messages between the file system agent instances, the cache manager and the volume manager. Any other type of message transport may be used as well.

[0028] The interface/filer server shown in Figure 4 may be implemented as a standalone system, or as a managed service. In the latter case, the system executes in an end user (local file system) environment. A managed service provider provides the system (and the versioned file system service), preferably on a fee or subscription basis, and the data store (the cloud) typically is provided by one or more third party service providers. The shared versioned file system may

have its own associated object-based data store, but this is not a requirement, as its main operation is to generate and manage the structured data representations that comprise the shared versioned file system. The cloud preferably is used just to store the structured data representations, preferably in a write-once manner, although the “shared versioned file system” as described herein may be used with any back-end data store and can be a write-many data store.

[0029] As described above, the file system agent 408 is capable of completely recovering from the cloud (or other store) the state of the local version of the shared versioned file system and providing immediate file system access (once FSA metadata is recovered). The FSA can also recover to any point-in-time for the whole shared versioned file system, a directory and all its contents, a portion of a directory (e.g., a shard) and its contents, a single file, or a piece of a file. These and other advantages are provided by the “shared versioned file system” of this disclosure, as it is now described in more detail below.

[0030] Figure 5 is a block diagram of the architecture of a shared versioned file system 50 according to an embodiment. The architecture 50 includes a root-level directory 500 and first-level directories 500-1, 500-2. First level directory 500-2 includes sub-directory 2-1 502, which is divided into shards 1, 2, and 3 (corresponding to reference numbers 503-1, 503-2, 503-3) (in general, shard 503). Each shard 503 is a portion of sub-directory 2-1 502. As an example, files 1, 2, and 3 (corresponding to reference numbers 504-1, 504-2, and 504-3) in sub-directory 2-1 502 are assigned to shard 1 503-1. Shard 2 503-2 and shard 3 503-3 can also include files and/or metadata that belong to sub-directory 2-1 502.

[0031] Each file 504 is divided into one or more chunks, such as chunks 1, 2, 3 (corresponding to reference numbers 504-1, 504-2, 504-3) (in general, chunk 504) of file 2 504-2. An example of dividing files into chunks can be found in U.S. Patent No. 8,566,362, entitled “Method and System for Versioned File System Using Structured Data Representations,” assigned to the present Applicant, which is incorporated herein by reference. Each directory/sub-directory, file, and chunk of shared versioned file system 50 can be represented by an inode. Example inode numbers for the following components of shared versioned file system 50 are illustrated in parentheses: sub-directory 2-1 502 (10), file 1 504-1 (101), file 2 504-2 (102), file 3 504-3 (103), and chunk 1 505-1 (1001). Additional inode numbers are illustrated in Figure 5.

[0032] Shard 503 can have an arbitrary number of files and/or metadata from sub-directory 2-1 502. In addition, or in the alternative, shard 503 can have a maximum number of files and/or metadata, for example to provide an increased size (horizontally and/or vertically) of the shared versioned file system.

[0033] Each shard 503 has a manifest that identifies the files (by inode number) assigned to that shard. For example, manifest 540 of shard 1 503-1 identifies inodes 101, 102, and 103. The manifest 540 also includes metadata about each inode, such as the version of the shard in which the inode (file) was created and the version of the shard in which the inode (file) was last modified. The manifest can also include a flag or bit to indicate whether any component of the shard has been modified, including the manifest itself.

[0034] In addition, each file 504 has a manifest that identifies the chunks (by inode number) that make up the data of the file. For example, manifest 550 of file 2 504-2 identifies inodes 1001, 1002, and 1003. The manifest also includes metadata about each inode, such as the relationship or offset between each inode. The manifest can also include a flag or bit to indicate whether any component of the file has been modified, including the manifest itself.

[0035] Figure 6 is a block diagram of the architecture of the shared versioned file system 50 after a change to chunk 1 505-1 in file 2 504-2. As illustrated by the asterisks in Figure 6, the change to chunk 1* 505-1 propagates to file 2* 504-2. In other words, the modification to chunk 1* 505-1 causes file 2* 504-2 to appear as modified or “dirty.” The modified or dirty file 2* 504-2 causes shard 1* 503-1 to appear as modified, which in turn causes sub-directory 2-1* 502 to appear as modified. The modification to chunk 1* 505-1 does not propagate past sub-directory 2-1* 502, such as to directory 2 501-2 or root 500. Thus directory 2 501-2 and root 500 appear as unmodified even if sub-directory 2-1* 502 appears as modified. In general, a change to any portion of the shared versioned file system 50 only propagates to the closest directory or sub-directory level. For example, a change to shard 2 503-2 propagates to sub-directory 2-1 502 but not to directory 2 501-2 or root 500. Similarly, a change to sub-directory 2-1 propagates to directory 2 501-2 but not to root 500.

[0036] By limiting the propagation of change events to the closest directory or sub-directory, shared versioned file system 50 can be synchronized more efficiently across local interfaces running respective local versions of the shared versioned file system.

[0037] As discussed above, a modification to a file or shard causes an update flag in the respective manifest to turn on, which makes the corresponding file or shard appear as modified. Using the example of Figure 6, the modification to chunk 1* 505-1 automatically causes the update flag in manifest* 550 to turn on, which in turn causes file 2* 504-2 to appear as modified. The modification to file 2* 504-2 causes the update flag in manifest* 540 to turn on, which in turn causes shard 1* 503-1 to appear as modified. When shard 1* 503-1 appears as modified, sub-directory 2-1* 502 also appears as modified since shard 1* 503-1 is a portion of sub-directory 2-1* 502.

[0038] Figure 7 is a block diagram of a system 70 for running a shared versioned file system according to an embodiment. The system 70 includes operations server 700, filer servers 710, 720, and user computers 712, 714, 722, 724. Filer servers 710, 720 can be the same as FSA 202 or FSA 306 described above. Each filer server provides a respective local version of the shared versioned file system to its respective user computers. For example, Filer server 710 exposes local version A 730 of the shared versioned file system to local computers 712, 714. Likewise, Filer server 720 exposes local version B 740 of the shared versioned file system to local computers 722, 724. Local version A 730 and local version B 740 can represent the same or different versions of the shared versioned file system based on how recently the respective filer server 710, 720 have retrieved updates to the shared versioned file system from operations server 700 and cloud storage 750. If filer servers 710, 720 have retrieved updates to the shared versioned file system up to the same change event (as discussed below), local versions 730, 740 of the shared versioned file system are identical. The filer servers 710, 720 can communicate with respective user computers over a network protocol such as Network File System (NFS), Server Message Block (SMB), or Common Internet File System (CIFS). In some embodiments, the operations server 700 is a NASUNI® Operations Center, available from Nasuni Corporation of Massachusetts.

[0039] In an example of the operation of system 70, a user on user computer 712 makes a modification to a document that corresponds to file 2 504-2 (using the example of Figures 5 and 6, discussed above). The modification occurs in the portion of file 2 504-2 corresponding to chunk 1 505-1. Filer server 710 saves a new version of File 2 504-2 locally. The new version of file 2 504-2 includes modified manifest 540* that contains modified chunk 1* 505-1 and pointers to unmodified chunk 2 505-2 and unmodified chunk n 505-n. Filer server 710 also saves a new

version of shard 1 503-1 locally. The new version (e.g., version 2) of shard 1 503-1 (i.e., modified shard 1* 503-1) includes a new manifest 550* that includes the the inode numbers of each file in modified shard 1* 503-1.

[0040] Continuing with the example of Figures 5 and 6, manifest 550* includes inodes 101 (unmodified file 1 504-1), 102 (modified file 2* 504-2), and 103 (unmodified file 3 504-3). In addition, manifest 550* indicates that inode 102 was last modified in version 2 of shard 1 (i.e., modified shard 1* 503-1). Manifest 550* also indicates that inodes 101 and 103 were last modified in version 1 of shard 1. Manifest 550* also turn the update flag on to indicate that modified shard 1* 503-1 contains at least one update. By comparing the present version of shard 1 (version 2) with the version number in which each inode was last modified (inode 101 (last modified in version 1), 102 (last modified in version 2), and 103 (last modified in version 1)), the filer server 710 can determine that inode 102 includes modified data while inodes 101 and 103 do not include modified data.

[0041] In another example, a user on computer 724 creates a new file called file 4 (inode 104) in shard 1 in the local version B of the shared versioned file system managed by filer server 720. The new manifest of shard 1 in local version B includes inodes 101 (unmodified file 1 504-1), 102 (unmodified file 2 504-2), 103 (unmodified file 3 504-3), and 104 (new file 4). The new manifest indicates that inodes 101-103 were each created in version 1 of shard 1 while inode 104 was created in version 2 of shard 1. The new manifest also includes a flag in the “on” state to indicate that version 2 of shard 1 contains at least one update. By comparing the present version of shard 1 (version 2) with the version number in which each inode was created (inode 101 (created in version 1), 102 (created in version 2), 103 (created in version 1), and 104 (created in version 2)), the filer server 720 can determine that inode 104 is new in version 2 of shard 1 while inodes 101-103 are not new.

[0042] Figure 8 is a flow chart 80 of a method for sending updated data to cloud storage according to an embodiment. Using the above example of creating a modified file 2* 504-2 in shard 1, in step 810 filer server 710 determines which directories or sub-directories have the updated flag flipped to the “on” state in local version A of the shared versioned file system 730. In the example of Figures 5 and 6, sub-directory 2-1 502 is the only directory or sub-directory in which the updated flag is flipped on. In step 820, filer server 710 determines which shards within

the updated directories/sub-directories identified in step 810 have the updated flag flipped to the on state. In the above example, shard 1 503-1 is the only shard in sub-directory 2-1 503-1 in which the updated flag is on. Since filer server 710 has at least one updated shard, the flow chart 80 proceeds to step 830. In the circumstance when there are no updated shards, the filer server would return to step 810. The filer server can wait for a short time period (e.g., 30 seconds to 1 minute) before returning to step 810.

[0043] In step 830, filer server 710 sends a request to operations server 700 for a global lock on shard 1 503-1. If a global lock is available and not in use by another interface or filer server, operations server 700 returns the global lock to Filer A 710. If the global lock is not available, operations server 700 returns a message to the filer server to indicate that the global lock is unavailable. In that case, the filer server 710 can request a global lock for another updated shard and request the global lock on shard 1 503-1 later. Alternatively, the filer server 710 can continue to request the global lock on shard 1 503-1 until the operations server 700 is able to provide it.

[0044] After filer server 710 receives the global lock, the flow chart 80 proceeds to step 840 in which case the filer server 710 identifies the portions of shard 1 503-1 that have updated information. This can be a query for the state of each shard directory entry in the cache of filer server 710 as described below. The available states are cache entry dirty (i.e., the shard directory entry contains updated information since the last shard version), cache entry clean (i.e., the shard directory entry does not contain updated information since the last shard version), or cache entry created (i.e., the shard directory entry did not exist in the last shard version; it was created in the present shard version). The shard directory entries of dirty and created contain new information and need to be sent to the cloud/data store. The shard directory entries of clean already exist in that form in the cloud/data store so the filer server does not need to send the clean entries to the cloud/data store. For each dirty entry, the filer server determines the portions of the directory entry (e.g., a chunk and/or a manifest of a file) that have been updated. In the example of Figures 5 and 6, the filer server 710 determines from the updated flags of files 1-3 (504-1 to 504-3) that file 2 504-2 is updated while file 1 504-1 and file 3 504-3 have not been updated. The filer server 710 then evaluates the manifest 550 of file 2 version 2 and determines the file version 2 includes chunk 1* 505-1 and pointers to chunk 2 505-2 and chunk 3 505-3. Based on this information, the

filer server 710 determines that chunk 1* 505-1 is new/updated and chunks 2 505-2 and 3 505-3 are not new.

[0045] Data is stored in cloud storage 750 by inode number and version number. For example, the contents of shard 1 503-1 in sub-directory 2-1 502 can be stored in the cloud at inodes/10/S1/now where “10” corresponds to the inode number for sub-directory 2-1 502, “S1” corresponds to shard 1 in inode 10 (sub-directory 2-1 502), and “now” is a pointer to the most recent version of shard 1. For example, if the most recent version of shard 1 is version 1 (i.e., now = 1), the pointer is to inodes/10/S1/v1. The directory inodes/10/S1/v1 includes pointers to the contents of shard 1 (i.e., inode 101 (file 1 504-1), inode 102 (file 2 504-2), and inode 1** (file n 504-n)). The pointer to each inode (file) is to the latest version of the inode (file). For example, inode 102 (file 2 504-2) includes a pointer to inodes/102/now. As before, “now” is a pointer to the most recent version, which in this case is the most recent version of inode 102. For example, if the most recent version of file 2 is version 3 (i.e., now = 3), the pointer is to inodes/102/3. Continuing with the illustration of Figure 5, the most recent version of file 2 includes a manifest 510 that identifies inode 1001 (chunk 1 505-1), inode 1002 (chunk 2 505-2), and inode 10** (chunk n 505n) and the relationship between the chunks (e.g., offset) as the components that form file 2.

[0046] Returning to the example above, in step 850 the filer server 710 sends the update portions of updated shard 1 to the cloud/data store. Filer server 710 can place a local lock on shard 1 during this step. First, filer server 710 creates a new version (version 2) on cloud storage for shard 1 503-1 at inodes/10/S1/v2. Version 2 of shard 1 includes a new manifest that identifies that the shard includes inodes 101-103 (corresponding to files 1-3). Since no files have been added or deleted from shard 1, the inodes identified in the manifest are the same in versions 1 and 2 of shard 1. However, the metadata for inode 102 indicates that inode 102 was created in version 1 of shard 1 and last updated in version 2 of shard 1. In contrast, the metadata for inodes 101 and 103 indicate that they were created in version 1 of shard 1 but have not been updated. Filer server 710 also updates the metadata for inodes/10/S1/now to reference version 2 of shard 1 as the latest version (i.e., now = 2).

[0047] To update the contents of inode 102 (file 2 504-2), filer server 710 creates a new version (version 2) at inodes/102/2. The most recent version of file 2 includes a new manifest

550 that identifies modified inode 1001 (chunk 1* 505-1) and pointers to unmodified inode 1002 (chunk 2 505-2) and unmodified inode 1003 (chunk 3 505-3) and the relationship between the chunks (e.g., offset) as the components that form version 2 of file 2 504-2. Filer A also updates the metadata for inodes/102/now to reference version 2 of file 2 as the latest version (i.e., now = 2). In addition, filer server 710 sends modified inode 1001 (chunk 1* 505-1) to the cloud/data store. When the update is complete, filer server 710 releases the global lock 860 on shard 1 503-1 back to operations server 700. Filer server 710 also releases the local lock on shard 1 503-1 if such a lock was placed on shard 1 503-1. In step 870, the filer server 710 determines if there are any additional updated shards that need to be sent to the cloud/data store. If so, the flow chart 80 returns to step 830 where the filer server 710 requests a global lock on the next updated shard. If there are no additional updated shards to send to the cloud/data store, the flow chart 80 returns to step 810 to re-start the cloud update process. The filer server 710 can wait for a predetermined time period (e.g., 1 to 5 minutes) before re-starting the flow chart 80.

[0048] As filer servers 710, 720 make updates to files and directories in the shared versioned file system, operations server 700 maintains a table 90 of such updates as illustrated in Figure 9. Table 90 includes the updated inode and the updated shard within the updated inode for each update. Table 90 also includes an event number that operations server 700 assigns to each update. Table 90 illustrates that the event number increases by one integer value for each update, though the event number can increase by a different amount in some embodiments. For example, the event number can increase by multiple integers, a decimal (e.g., 100.1, 100.2, etc.), or other unit. The update to shard 1 of inode 10 (sub-directory 2-1 502) described above is included as event number 102 in table 90.

[0049] Filer servers 710, 720 query the operations server 700 periodically to determine whether there are any recent updates to the shared versioned file system as indicated by the event number. For example, filer server 720 last synchronized updates to the shared versioned file system at event number 100 as illustrated in Figure 8. Since that time, there have been 5 updates to the shared versioned file system, as represented by event numbers 101-105. In order for filer server 720 to update its local version 740 of the global file system with the latest changes, filer server 720 retrieves and merges the updates represented by event numbers 101-105 into its local version 740 of the global file system.

[0050] Likewise, filer server 710 last synchronized updates to the shared versioned file system at event number 102, the same event that filer sever 710 updated shard 1 of inode 10 (sub-directory 2-1 502), as described above. To update its local version 730 of the global file system with the latest changes, filer server 710 retrieves and merges the updates represented by event numbers 103-105 into its local version 730 of the global file system.

[0051] Figure 10 is a flow chart 1000 of a method for synchronizing updates from a local version of a shared versioned file system to a cloud data store according to an embodiment. In step 1010, the operations server receives a request for a global lock on a shard, such as shard 1 of inode 10. At step 1020, the operations server determines if the global lock is available for the requested shard. If the global lock is available and not in use by another filer server, the operations server sends the global lock to the requesting filer server in step 1030. If the global lock is not available, the operations server can continue to check if for the global lock in an available state. In addition or in the alternative, the operations server can respond to the filer server that the global lock is not available. The filer server can optionally repeat the request for the global lock on the requested shard.

[0052] After sending the global lock to the requesting filer server in step 1030, the operations server adds a new event to the update table in step 1040. The update table can be the same or substantially the same as the table illustrated in Figure 9. In general, the update table is a list of each update to a shard in the cloud-based data store. Each update is assigned an event number. The update table can be used by the filer servers to synchronize updates from the cloud-based data store to their respective local versions of the shared versioned operating system. After the requesting filer server has pushed the update directory entries of the requested shard to the cloud-based data store, the operations server receives 1050 the global lock back from the requesting filer server.

[0053] Figure 11 is a flow chart 1100 of a method for synchronizing updates from a cloud-based a shared versioned file system to a local version of same according to an embodiment. In step 1110, the filer server queries the operating sever for a list of updates to the global file system that have occurred since the last event number updated to the file server. As an example with respect to Figure 9, Filer 720 queries the operating server for a list of updates that have occurred since event 100, the last event number updated to file server 720. In the query, the

file server can include the last event number updated to the file server in which case the operating server determines if the file server has the most recent updates by comparing the last event number updated to the file server with the most recent event number on the operations center. Alternatively, the file server rver can request the operations server for the most recent event number and the file server r can compare the last event number updated to the file server with the most recent event number on the operations server.

[0054] In step 1120, the file server or operations server determines if there are any new (unsynchronized) event numbers on the operations server. If the query in step 1010 includes the last event number updated to the file server, the operations center compares the last event number and the most recent event number to determine if there are any new events. Alternatively, if the file server requested the operations server for the most recent event number (and did not send the last event number in the query), the file server determines if there are any updates by comparing the most recent event on the operations server with the last event number updated to the file server, as discussed above. If there are new events, the file server requests the operations center to provide the inode number and shard number associated with each new event number.

[0055] If the result of step 1120 is that there are no new events since the last event number, the flow chart 1100 returns to step 1110. In some embodiments, the file server briefly pauses (e.g., for 30 seconds to 1 minute) before returning to step 1110.

[0056] If the result of step 1120 is that there are new events since the last event number, the flow chart 80 proceeds to step 1130. In step 1130, the file server receives, for each new event, the inode number and shard number associated with the new event. Using the example of Figures 6 and 7 above, the new event includes inode 10 (sub-directory 2-1 502) and shard 1 (e.g., in the form of /inode/10/s1).

[0057] In step 1140, the file server retrieves the latest version of each shard received from the operations server in step 1130. As discussed above, each shard includes a manifest of its shard directory entries (e.g., inodes corresponding to files) and metadata about each shard directory entry, such as the version of the shard in which it a file (inode) was created and the version of the shard in which the file (inode) was last updated. The file server uses this metadata in steps 1150 and 1160 to determine the state of each directory entry in the latest cloud version of

the shard (step 1150) and the state of each directory entry in the cache version of the shard (step 1160). In step 1170, the file server performs the appropriate operation on each cache directory entry according to the table below. In step 1180, the file server determines if there are any additional updated shards received from the operations center that have not been processed. If so, the file server returns to step 1150 to determine the state of each directory entry in the next unprocessed shard. This loop continues until all updated shards received from the operations center have been processed. After all updated shards received from the operations center have been processed, the filer server in step 1180 returns to step 1110 to query the operation server for updates since the last event number. In this case, the last event event number updated to the filer server would be the last event number from step 1130 in the last iteration through flow chart 1100.

[0058] The state of a given entry in a cloud shard version can be determined as follows.

[0059] If the version number in which a directory entry (e.g., File 1) in cloud shard 1 (a representative shard number) was last modified is the same as the latest version number of cloud shard 1, this indicates that File 1 was updated or modified (in general, “dirty”) in the latest version of cloud shard 1. In other words, the new event for shard 1 was due, at least in part, to an update or modification to File 1. As a shorthand, this state is referred to as “cloud entry dirty.”

[0060] If the version number in which File 1 in cloud shard 1 was last modified is the less than the latest version number of cloud shard 1, this indicates that File 1 was not updated or modified in the latest version of cloud shard 1. In other words, the new event for shard 1 was not due to File 1. As a shorthand, this state is referred to as “cloud entry clean.”

[0061] If the version number in which File 1 in cloud shard 1 was created is the same as the latest version number of cloud shard 1, this indicates that File 1 was created in the latest version of cloud shard 1. In other words, the new event for shard 1 was due, at least in part, to the creation of File 1. As a shorthand, this state is referred to as “cloud entry created.”

[0062] If File 1 is not found in the latest version of cloud shard 1, this indicates that File 1 does not exist in that version. For example, this would occur if a user deleted File 1 and the filer server pushed cache shard 1 with the deleted file to the cloud. As a shorthand, this state is referred to as “cloud entry not found.”

[0063] The state of a given entry in a cache shard version can be determined as follows.

[0064] If the version number in which File 1 in cache shard 1 was last modified is different than the latest version number of cache shard 1, this indicates that File 1 has been updated or modified (in general, “dirty”) since the filer server retrieved the latest cloud shard version from the cloud and merged it into local cache. In other words, cache shard 1 includes at least one modified directory entry that needs to be pushed to the cloud, at which point a new event number will be created at the operations center. As a shorthand, this state is referred to as “cache entry dirty.”

[0065] If the version number in which File 1 in cache shard 1 was last modified is the same as the latest version number of cache shard 1, this indicates that File 1 has been not been updated or modified since the filer server retrieved the latest cloud shard version from the cloud and merged it into local cache. As a shorthand, this state is referred to as “cache entry clean.”

[0066] If the version number in which File 1 in cache shard 1 was created is different than the latest version number of cache shard 1, this indicates that File 1 was created since the filer server retrieved the latest cloud shard version from the cloud and merged it into local cache. As a shorthand, this state is referred to as “cache entry created.”

[0067] If File 1 is not found in the latest version of cache shard 1, this indicates that File 1 does not exist in that version. For example, this would occur if a user deleted File 1 after the filer server retrieved the latest cloud shard version from the cloud and merged it into local cache. As a shorthand, this state is referred to as “cache entry not found.”

[0068] The filer server performs different operations depending on the state of a directory entry (e.g., File 1) in the cloud shard and in the cache shard. These operations are summarized in Table 1 and described below. The description below continues to use File 1 and shard 1 as a representative directory entry and shard for discussion purposes.

Cloud Entry State	Cache Entry State	Operation
Created	Clean	N/A
Created	Dirty	Conflict

Created	Not Found	Create cache entry
Created	Created	Conflict
Dirty	Clean	Update cache entry
Dirty	Dirty	Conflict
Dirty	Not Found	N/A
Dirty	Created	Conflict
Clean	Clean	Keep cache entry
Clean	Dirty	Keep cache entry
Clean	Not Found	N/A
Clean	Created	Keep cache entry
Not Found	Clean	Delete cache entry
Not Found	Dirty	Keep cache entry
Not Found	Created	Keep cache entry

Table 1

[0069] If the state of File 1 is created in cloud shard 1 and it is clean in cache shard 1, the filer server determines it is not an applicable state and returns an error. This is indicative of a coding error as such a combination is not possible.

[0070] If the state of File 1 is created in cloud shard 1 and it is dirty in cache shard 1, the filer server determines that there is a conflict. When a conflict occurs, the filer server saves the conflicted File 1 in cache shard 1 to the cloud and changes the file name to indicate that it is a conflicted file (e.g., File 1_conflicted).

[0071] If the state of File 1 is created in cloud shard 1 and it is not found in cache shard 1, the filer server creates a copy of File 1 in a new version of cache shard 1.

[0072] If the state of File 1 is created in cloud shard 1 and it is also created in cache shard 1, the filer server determines that there is a conflict. This scenario could occur if users associated with different filer server create a file with the same name in the same directory (shard). In a

conflict state, the filer server saves conflicted version of File 1 from cache shard 1 to the cloud and changes its file name to indicate that it is a conflicted file, as described above.

[0073] If the state of File 1 is dirty in cloud shard 1 and it is clean in cache shard 1, the filer server merges the updates from the cloud version of File 1 into the cache version of File 1, as discussed herein. This scenario could occur if a user associated with filer server A makes an update to File 1 and sends that update to the cloud while filer server B has a clean copy in cache of the prior version of File 1. Thus filer server B has an old version of File 1 and needs to synchronize with the cloud to obtain the updates to File 1.

[0074] If the state of File 1 is dirty in cloud shard 1 and it is dirty in cache shard 1, the filer server determines that there is a conflict and proceeds as described above. This scenario could occur if two users make an update to the same version of File 1 close in time to one another. For example, a user associated with filer server A makes an update to File 1 and sends that update to the cloud while a second user associated with filer server B also makes an update to the same version of File 1, but has not yet pushed that update to the cloud.

[0075] If the state of File 1 is dirty in cloud shard 1 and it is not found in cache shard 1, the filer server determines it is not an applicable state and returns an error. This is indicative of a coding error as such a combination is not possible.

[0076] If the state of File 1 is dirty in cloud shard 1 and it is created in cache shard 1, the filer server determines that there is a conflict. This scenario could occur if a user associated with filer server A makes an update to File 1, which already exists in the cloud while a user associated with filer server B deletes File 1 and then creates a new File 1. The filer server saves conflicted cache version of File 1 in shard 1 to the cloud and changes its file name to indicate that it is a conflicted file, as described above.

[0077] If the state of File 1 is clean in cloud shard 1 and it is clean in cache shard 1, the filer server keeps the cache version of File 1 since there have been no changes to the file.

[0078] If the state of File 1 is clean in cloud shard 1 and it is dirty in cache shard 1, the filer server keeps the cache version of File 1. The filer server will merge the updates to File 1 the next time that the filer server pushes its updates or snapshot to the cloud. This scenario could

occur if the filer server has a modified version of File 1 in cache but has not yet pushed the new version of File 1 to the cloud.

[0079] If the state of File 1 is clean in cloud shard 1 and it is not found in cache shard 1, the filer server determines it is not an applicable state and returns an error. This is indicative of a coding error as such a combination is not possible.

[0080] If the state of File 1 is clean in cloud shard 1 and it is created in cache shard 1, the filer server keeps the cache version of File 1. The filer server will merge the updates to File 1 the next time that the filer server pushes its updates or snapshot to the cloud.

[0081] If the state of File 1 is not found in cloud shard 1 and it is clean in cache shard 1, the filer server deletes the cache version of File 1. This scenario could occur if a user has deleted File 1 and pushed that deletion to the cloud, but another user (associated with another filer server) has a prior version of shard 1 in which File 1 is clean.

[0082] If the state of File 1 is not found in cloud shard 1 and it is dirty in cache shard 1, the filer server keeps the cache version of File 1. This scenario could occur if a user associated with filer server A deletes File 1 and pushes that update to the cloud while a user associated with filer server B updates File 1. The updated version of File 1 will be sent to the cloud the next time filer server B pushes its updates/snapshot to the cloud.

[0083] If the state of File 1 is not found in cloud shard 1 and it is created in cache shard 1, the filer server keeps the cache version of File 1. This scenario could occur if a user creates a file that does not yet exist in the cloud. File 1 will be sent to the cloud the next time the filer server pushes its updates/snapshot to the cloud.

[0084] Figure 12 is a simplified illustration of a representative shard according to an embodiment. The representative shard in Figure 12 is shard 1 (i.e., S1) of inode 1, which is illustrated in the format of /inodes/[inode number]/[shard number]/[shard version number]. Using this format, version 1 of shard 1 in inode 1 is represented as /inodes/1/S1/1. As described above, the latest version number of a shard or inode can be located in cloud storage by the version number “now.” The “now” version subdirectory includes a pointer to the latest version, which in this case is version 1 (i.e., “latest” = “1”). Figure 12 illustrates the manifest 1200 of shard 1 version 1, which is written in XML (though other hierarchical coding languages can be

used). The manifest identifies its inode and shard number using respective <inode> and <shard> tags. The manifest also includes a list of directory entries in shard 1 version 1. In this example, the only directory entry is for inode 100, which has the name of file1.txt. The manifest also indicates that inode 100 has a size of 1,024 bytes.

[0085] Figure 13 is a simplified illustration of the representative shard from Figure 12 after an update according to an embodiment. As illustrated in Figure 13, the “latest” metadata has been updated with a pointer to version 2 of shard 1 of inode 1 (i.e., “latest” = “2”). In manifest 1300 of shard 1 version 2, it is apparent that inode 101 (file 2.txt) has been added to shard 1. Inode 101 has a size of 2,048 bytes. Thus, manifest 1300 includes the directory entries of inode 100 (file1.txt) and inode 101 (file2.txt).

[0086] Figure 14 is a simplified illustration of a representative directory entry in the representative shard of Figure 13. The representative directory entry in Figure 14 is inode 101, which corresponds to file2.txt as discussed above. The directory entry is illustrated in the format of /inodes/[inode number]/[inode version number]. Using this format, version 1 of inode 101 is represented as /inodes/101/1. As described above, the latest version number of a shard or inode can be located in cloud storage by the version number “now.” The “now” version subdirectory includes a pointer to the latest version, which in this case is version 1 (i.e., “latest” = “1”). The manifest 1400 identifies its inode number and the chunks that form the inode. In this case, the manifest 1400 indicates that inode 101 is formed of chunks having a handle (or name) of c1 and c2. The manifest 1400 also includes metadata on the relationship between the chunks. In this case, manifest 1400 indicates that chunk c1 has an offset of 0 and a length of 1,024 bytes. Manifest 1400 also indicates that chunk c2 has an offset of 1,024 and a length of 1024 bytes. In other words, inode 101 has a total length of 2,048 bytes where chunk c1 precedes chunk c2.

[0087] Chunks c1 and c2 each refer to an object in the cloud object store. In particular, chunk c1 refers to the directory /chunks/c1/data which includes a pointer to the latest version of chunk c1, which in this case is version 1. Thus, version 1 of chunk 1 can be found at /chunks/c1/refs/100/1. Likewise chunk c2 refers to the directory /chunks/c2/data which includes a pointer to the latest version of chunk c2, which in this case is version 1. Thus, version 1 of chunk 2 can be found at /chunks/c2/refs/100/1.

[0088] In the approach described above, the inode-based approach enables file versioning to the cloud. In this approach, and when a file is changed, it is not necessary to create a new version of the entire local file system let alone at a snapshot period; indeed, upon a given occurrence in the local file system with respect to the file, a new version of just the file is created for export to the cloud, and there is no longer any requirement for the system to wait on a “snapshot” to do so. To this end, and as has been described, the system is structured as an inode-based file system. File system objects are indexed, stored and retrieved in cloud storage by a globally-unique inode. With an inode-based approach, the hierarchy of the file system need not be maintained.

[0089] According to this technique, all versions, directories and files are referenced by inode. The following provides a description of a representative cloud storage layout.

[0090] Preferably, a file manifest for a given inode is stored by (inode, version) at a cloud path: /inodes/INODENUMBER/VERSION. A directory manifest for a given inode is stored by (inode, shard, version) at a cloud path: inodes/INODENUMBER/SHARDNUMBER/VERSION. A cloud metadata tag “latestversion” at /inodes/INODENUMBER/now (or ... /SHARDNUMBER/now) points to a latest manifest. In the event of cloud metadata loss, a new version is re-built given a cloud LIST of /inodes/INODENUMBER (or .../SHARDNUMBER). A special “manifest version 0” file (a version index) is stored at /inodes/INODENUMBER/index.

[0091] Preferably, file chunks are encrypted sections of a file, and they can be shared by any number of file manifest versions in the cloud filesystem. File chunk data for “chunkN” is stored at cloud path: /chunks/chunkN/data. File chunk reference by (inode, version) are stored at cloud path: /chunks/chunkN/inode/version.

[0092] For each file, a file manifest is created. Preferably, it is an XML document that describes the file inode, version and cloud chunks references (offset, length, handle).

[0093] For directories, a directory may comprise a set of directory shards, each representing a smaller independent piece of the directory. The latest shard version for a given inode number preferably is stored as a “latest” metadata tag on path: /inode/INODENUMBER/SHARDNUMBER/now. A directory shard manifest is an XML document that describes each directory entry in the shard, inode number, stbuf, and xattrs.

Preferably, stbuf, size, and attrs are stored inside a direntry for a given directory entry, other approaches may be used, e.g., large xattrs may be overflowed to a file manifest object.

[0094] FIG. 15 depicts a cloud storage layout example using the above-described data structures. This example depicts a portion of a directory manifest 1500, a portion of a file manifest 1502, and several inode lookup tables, namely, a directory inode table 1504, a file inode table 1506, and a chunk inode table 1508. More generally, the tables are data structures in the system. In this example scenario, and on initial format, the cloud contains “/inodes/1/now”, “/inodes/1/” and “inodes/1/S1.” The cloud metadata on “/inodes/1/now” points to 1. Now, assume two (2) files are created locally as follows: Initial format/(1); Create file /file1.text (100) size 2048; Create file /file2.text (101) size 2048. The numbers 100 and 101 represent inodes. The files are exported to the cloud (as part of shard S1), which then updates the cloud storage layout as depicted. Because shard S1 in the directory has been updated, the topmost entry in a directory inode table 1504 now reads as /inodes/1/S1/now (“latest” = “2”). As shown, directory manifest 1500 includes shard S1 with two directory entries for the inodes 100 and 101. Each directory entry includes the information for the file that has been created (updated or otherwise modified) locally. The inode in the directory manifest (e.g., inode 100) points to the file inode table 1506, which in this example has been updated to include the topmost entry: /inodes/100/now (“latest” = “1”), now the latest version of the file with that inode number. The file manifest 1502 includes the data identifying, for the particular, inode 100 (corresponding to the inode tag), the identity of the one or more chunks corresponding to the file. The chunks each having a corresponding handle. Chunk inode table 1508 then references each chunk, pointing to the inode and the version number. This de-referencing approach obviates any requirement to propagate changes to a root of a file system version, as files are each independently versioned (as opposed to a file system itself being versioned). The approach also enables support for files irrespective of their size, and it enables the cloud storage to send and receive changes even at a sub-file level. File version storage in this manner also provides for quicker response times, more efficient use of bandwidth, and improved file synchronization.

[0095] The techniques improve an operation of a write-once object store, which previously could not support an inode-based storage system that conventionally requires rewrite-in-place functionality. By associating a set of inode versions of an inode with the special de-referencing pointer (and managing those inode versions) as has been described, the techniques

herein enable the write-once object store to be transformed into a network-accessible file server. File changes that occur locally are then saved to the cloud using the cloud storage that is uniquely configured to support inode-based storage in the manner described.

[0096] It is not required that every inode version point to a unique copy of every chunk. If a particular chunk of a file has not changed from version 1 to version 2, that same single chunk is used by the inode at both version 1 and version 2.

[0097] While the disclosed subject matter has been described in the context of a method or process, the subject matter also relates to apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may comprise a computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable storage medium, such as, but is not limited to, any type of disk including an optical disk, a CD-ROM, and a magnetic-optical disk, a read-only memory (ROM), a random access memory (RAM), a magnetic or optical card, or any type of media suitable for storing electronic instructions, and each coupled to a computer system bus. A computer-readable medium having instructions stored thereon to perform the interface functions is tangible.

[0098] A given implementation of the disclosed subject matter is software written in a given programming language that runs on a server on a hardware platform running an operating system such as Linux. As noted above, the interface may be implemented as well as a virtual machine or appliance, or in any other tangible manner.

[0099] While given components of the system have been described separately, one of ordinary skill will appreciate that some of the functions may be combined or shared in given instructions, program sequences, code portions, and the like.

[00100] It is appreciated that certain features of the disclosed subject matter, which are, for clarity, described in the context of separate embodiments, may also be provided in combination in a single embodiment. Conversely, various features of the disclosed subject matter that are, for brevity, described in the context of a single embodiment, may also be provided separately or in any suitable sub-combination.

[00101] Unless otherwise defined, all technical and scientific terms used herein have the same meanings as are commonly understood by one of ordinary skill in the art to which this invention belongs. Although methods similar or equivalent to those described herein can be used in the practice or testing of the present invention, suitable methods are described herein. The present materials, methods, and examples are illustrative only and not intended to be limiting.

[00102] It will be appreciated by persons skilled in the art that the subject matter herein is not limited to what has been particularly shown and described hereinabove. Rather the scope of the subject matter is defined by the appended claims and includes both combinations and sub-combinations of the various features described hereinabove as well as variations and modifications thereof, which would occur to persons skilled in the art upon reading the foregoing description.

Claims

1. A method for synchronizing updates of a shared versioned file system to a local filer server comprising a processor, the method comprising:

in the local filer server:

maintaining at least a portion of a local version of the shared versioned file system in a cache of the local filer server, the shared versioned file system having a subdirectory divided into a plurality of shards;

sending a request to an operations server for updates to the shared versioned file system, the shared versioned file system stored in a data store in a cloud, the request including a stored event reference number that corresponds to a last shard synchronized from the data store to the local filer server;

when the stored event number is less than a current event reference number on the operations server, receiving an identity of at least one recent shard update in the cloud data store that has occurred since the stored event reference number; and

retrieving the at least one recent shard update from the cloud data store.

2. The method of claim 1, further comprising, for each recent shard update, merging the recent shard update into a corresponding local shard stored in the cache of the local filer server.

3. The method of claim 1, further comprising applying a local lock on the corresponding local shard while merging the recent shard update into the corresponding local shard.

4. The method of claim 1, further comprising determining a cloud state of each cloud shard directory entry in the recent shard update.

5. The method of claim 4, further comprising determining a cache state of each local shard directory entry in the corresponding local shard.

6. The method of claim 5, further comprising, if the cloud state of a first cloud directory entry is dirty and the cache state of a corresponding first cache directory entry is clean, merging the first cloud directory entry into the corresponding first directory cache entry.

7. The method of claim 6, wherein the first cloud directory entry is an updated cloud file and the corresponding first cache directory entry is a cache file.

8. The method of claim 7, wherein merging comprises updating a cache manifest of the cache file with a pointer to an unmodified chunk of the cache file and with a new inode version number corresponding to a modified chunk of the cache file.

9. The method of claim 7, wherein merging comprises updating a cache manifest of the cache file with a pointer to an unmodified chunk of the cache file and with a new inode number corresponding to a new chunk of the cache file.

10. The method of claim 5, further comprising, if the cloud state of a first cloud directory entry is created and the cache state of a corresponding first cache directory entry is not found, adding the first cloud directory entry to the corresponding local shard.

11. The method of claim 1, further comprising receiving the current event number from the operations server and storing the current event number as the stored event number.

12. A method for synchronizing updates of a shared versioned file system, the method comprising:

in an operations server in communication with a local filer server running a local version of the shared versioned file system, the operations server and the local filer server each comprising a processor, the shared versioned file system stored in a cloud-based data store:

receiving a request from the local filer server for updates to the shared versioned file system, the request including a stored event number that corresponds to a last shard synchronized from the cloud-based data store to the local filer server;

comparing the stored event number with a current event number, the current event number representing a latest shard updated in the cloud-based data store; and

when the stored event number is less than the current event number, sending an identity of each shard that has been updated in the cloud-based data store since the stored event number.

13. The method of claim 12, further comprising if the stored event number is equal to the current event number, sending a message to the local filer server that indicates that there are no updates to the shared versioned file system since the stored event number.

14. The method of claim 12, wherein the identity of each shard includes a directory or subdirectory and a shard number in the shared versioned file system that includes the update.

15. The method of claim 14, wherein the directory or subdirectory is identified by an inode number.

16. A filer server for synchronizing updates of a shared versioned file system stored in a cloud-based data store to a local version of the shared versioned file system, the filer server comprising:

a processor;

a cache memory that stores at least a portion of the local version of the shared versioned file system;

a computer memory storing computer program instructions executed by the processor to:

send a request to an operations server for updates to the shared versioned file system, the shared versioned file system stored in a data store in a cloud, the request including a stored event number that corresponds to a last shard synchronized from the data store to the local filer server;

when the stored event number is less than a current event number on the operations server, receive an identity of at least one recent shard update in the cloud data store that has occurred since the stored event number; and

retrieve the at least one cloud recent shard update from the cloud data store.

17. An operations server comprising:

a processor;

a computer memory storing computer program instructions executed by the processor to:

receive a request from a local filer server for updates to a shared versioned file system, the request including a stored event number that corresponds to a last shard synchronized from a cloud-based data store to the local filer server;

compare the stored event number with a current event number, the current event number representing a latest shard updated in the cloud-based data store; and

when the stored event number is less than the current event number, send an identity of each shard that has been updated in the cloud-based data store since the stored event number.

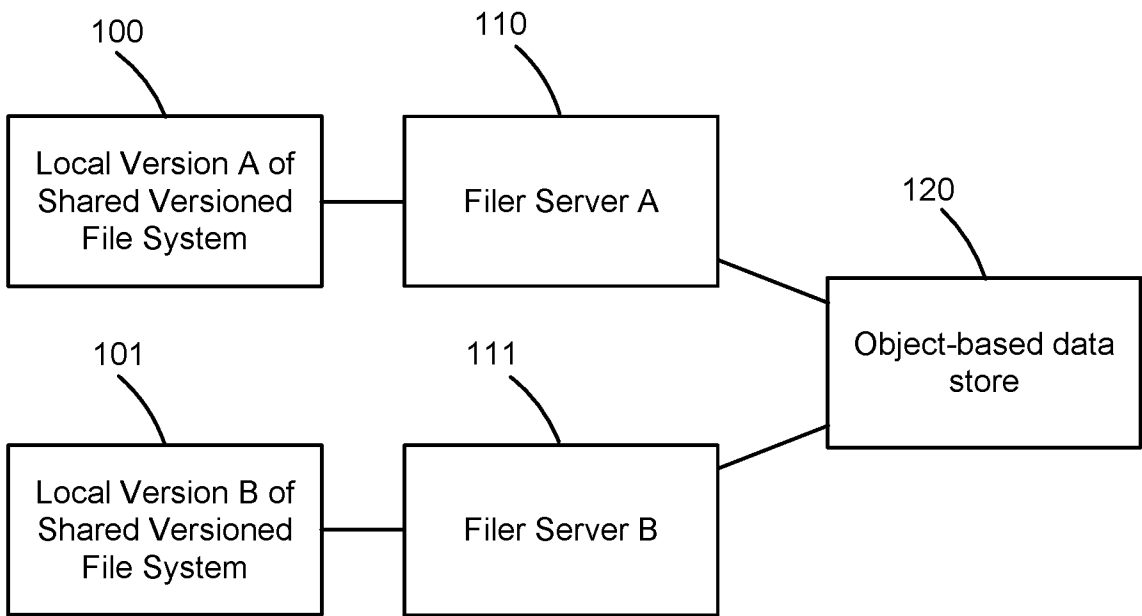


Fig. 1

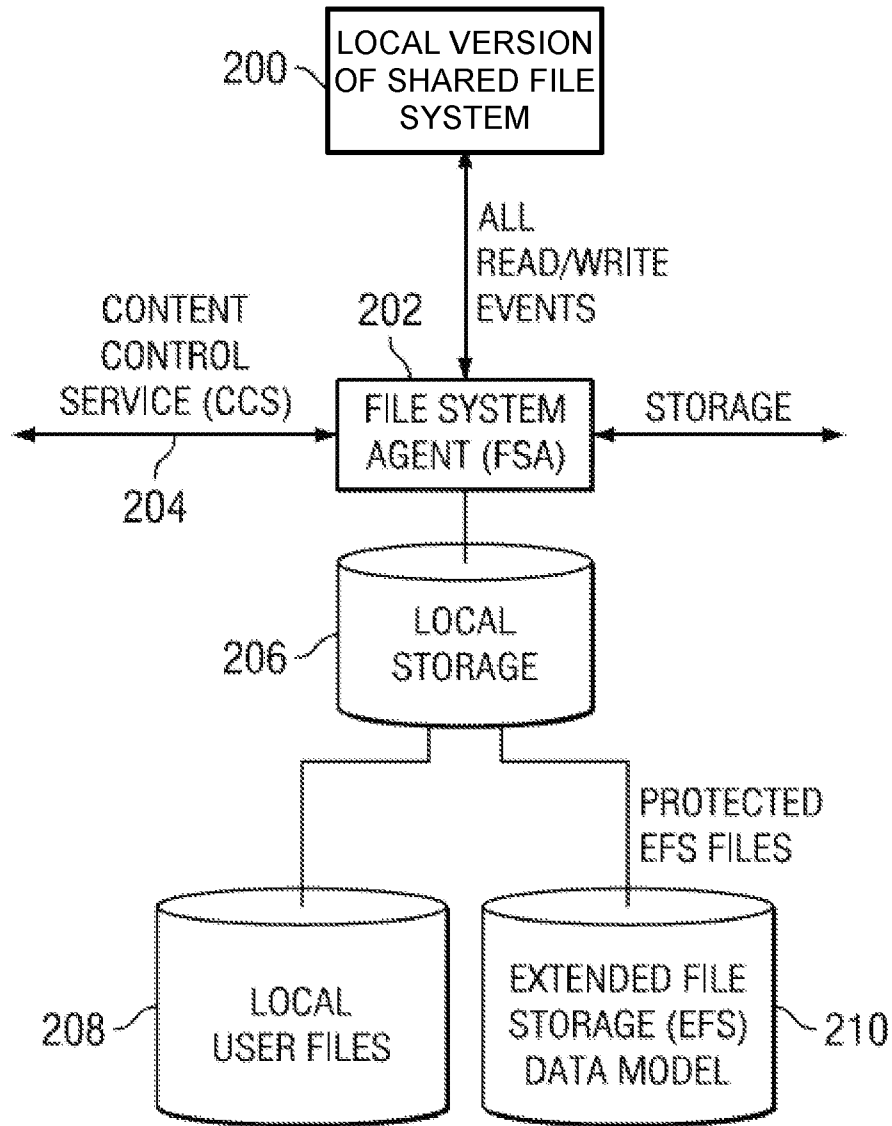


Fig. 2

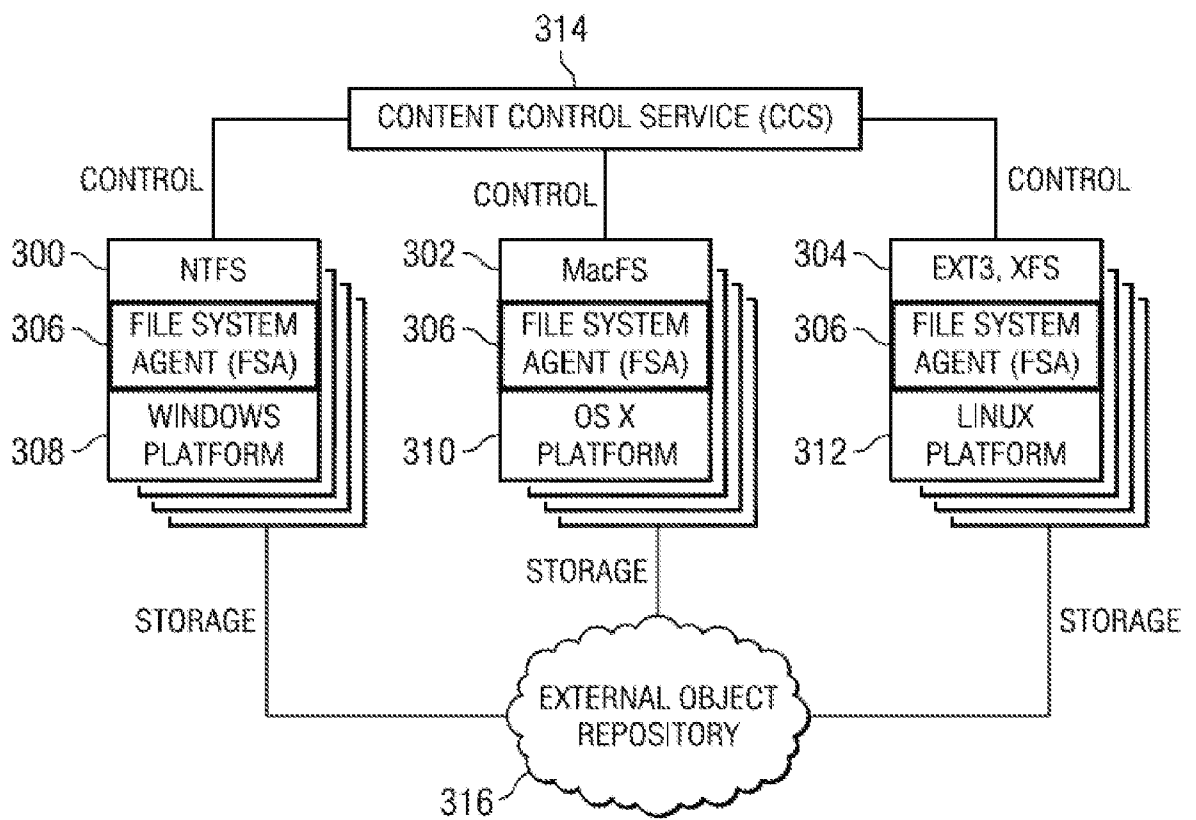


Fig. 3

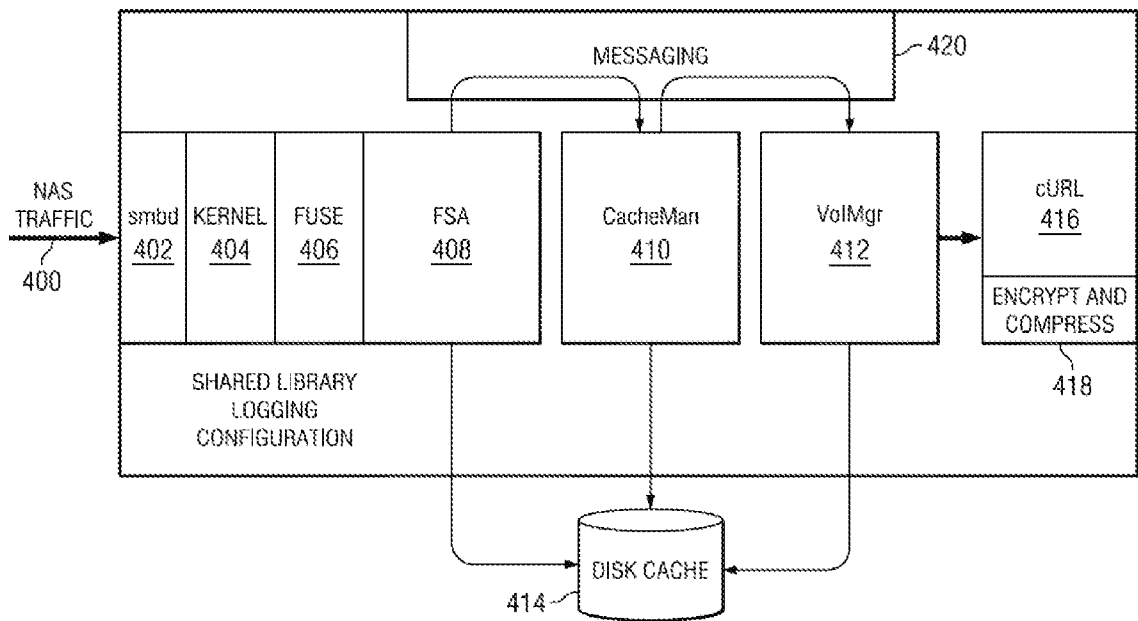


Fig. 4

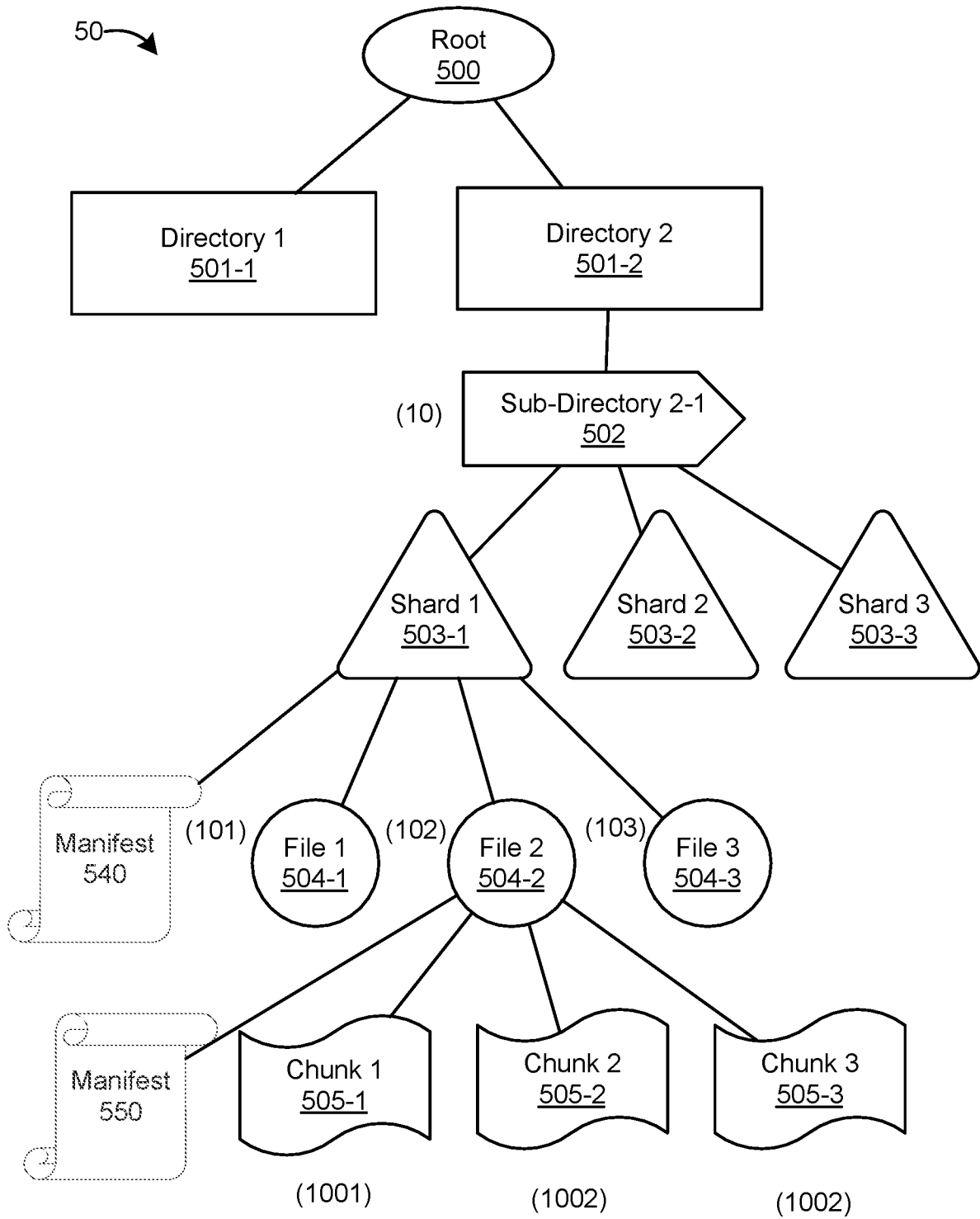


Fig. 5

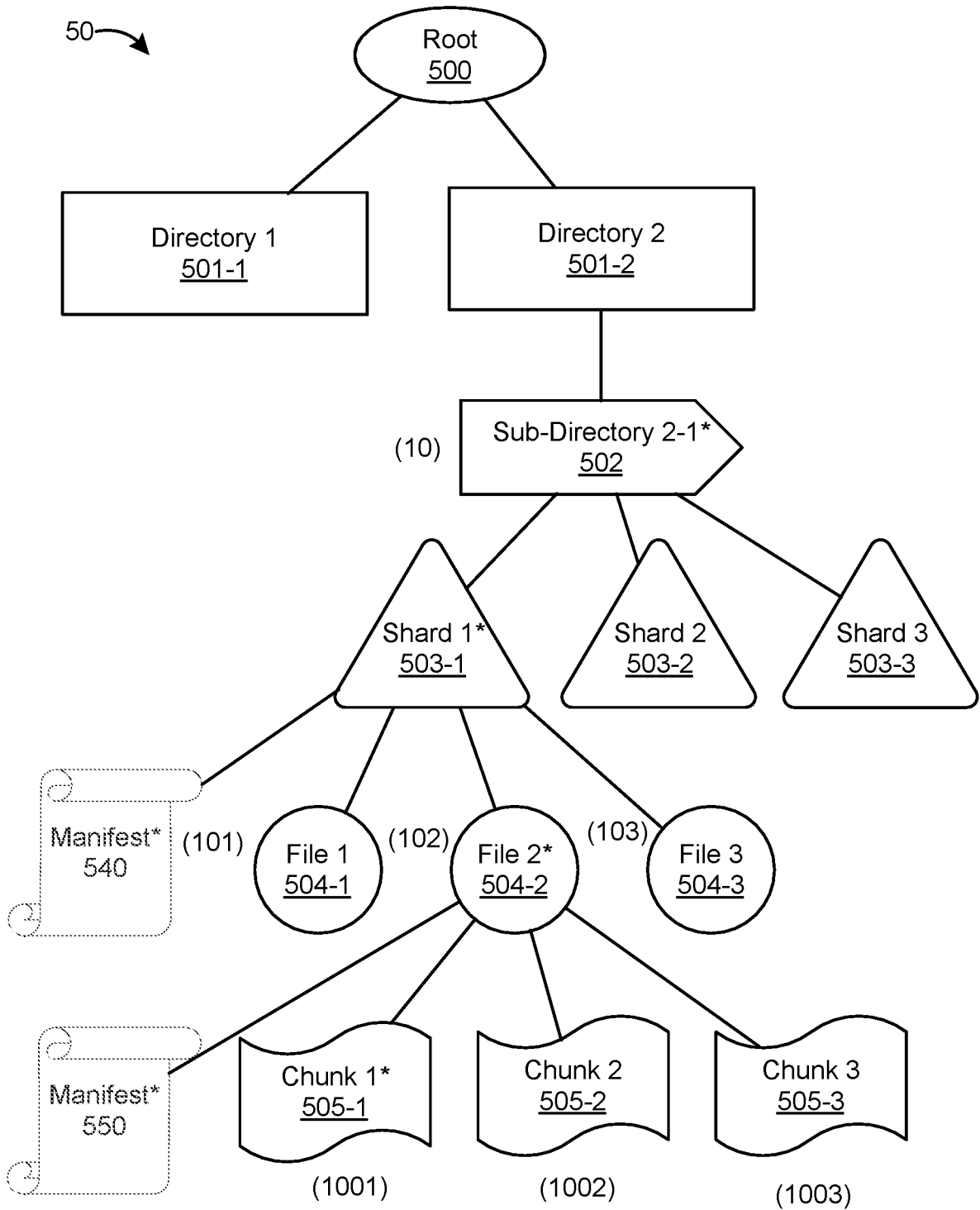


Fig. 6

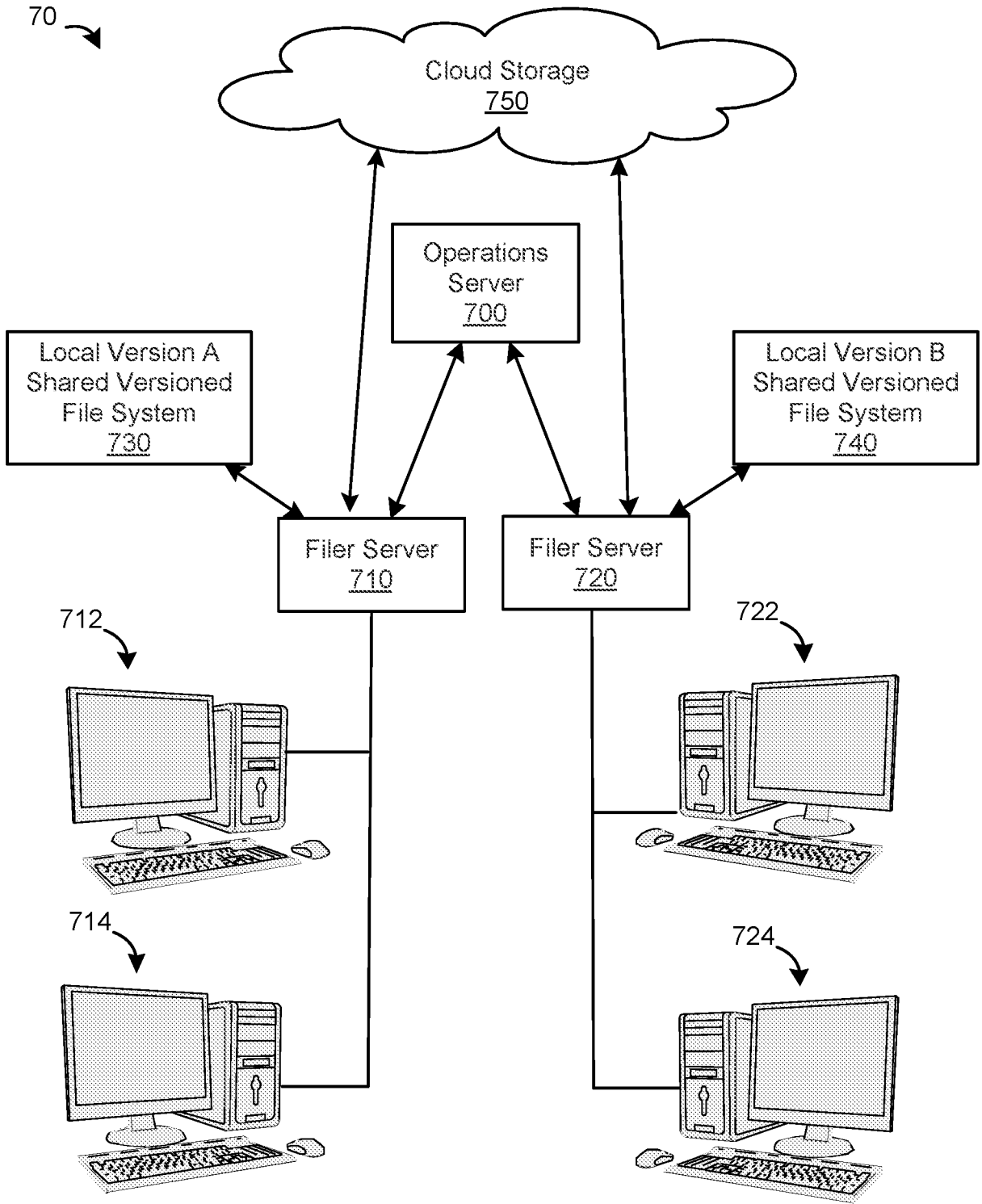


Fig. 7

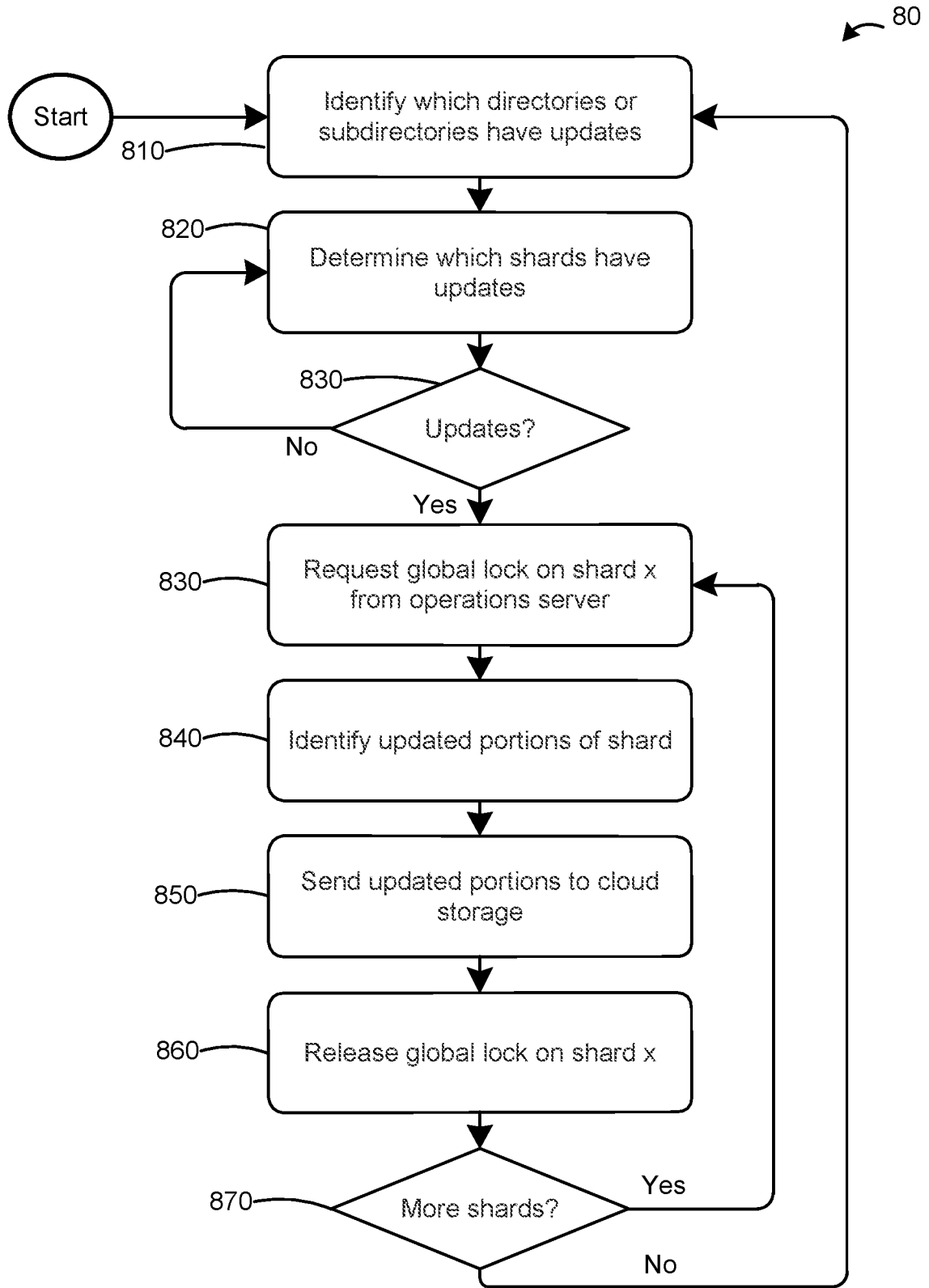


Fig. 8

90 ↘

<u>Inode</u>	<u>Shard</u>	<u>Update?</u>	<u>Event Number</u>
2	S1	Yes	100 ← Filer 720
4	S2	Yes	101
10	S1	Yes	102 ← Filer 710
10	S3	Yes	103
250	S5	Yes	104
150	S2	Yes	105

Fig. 9

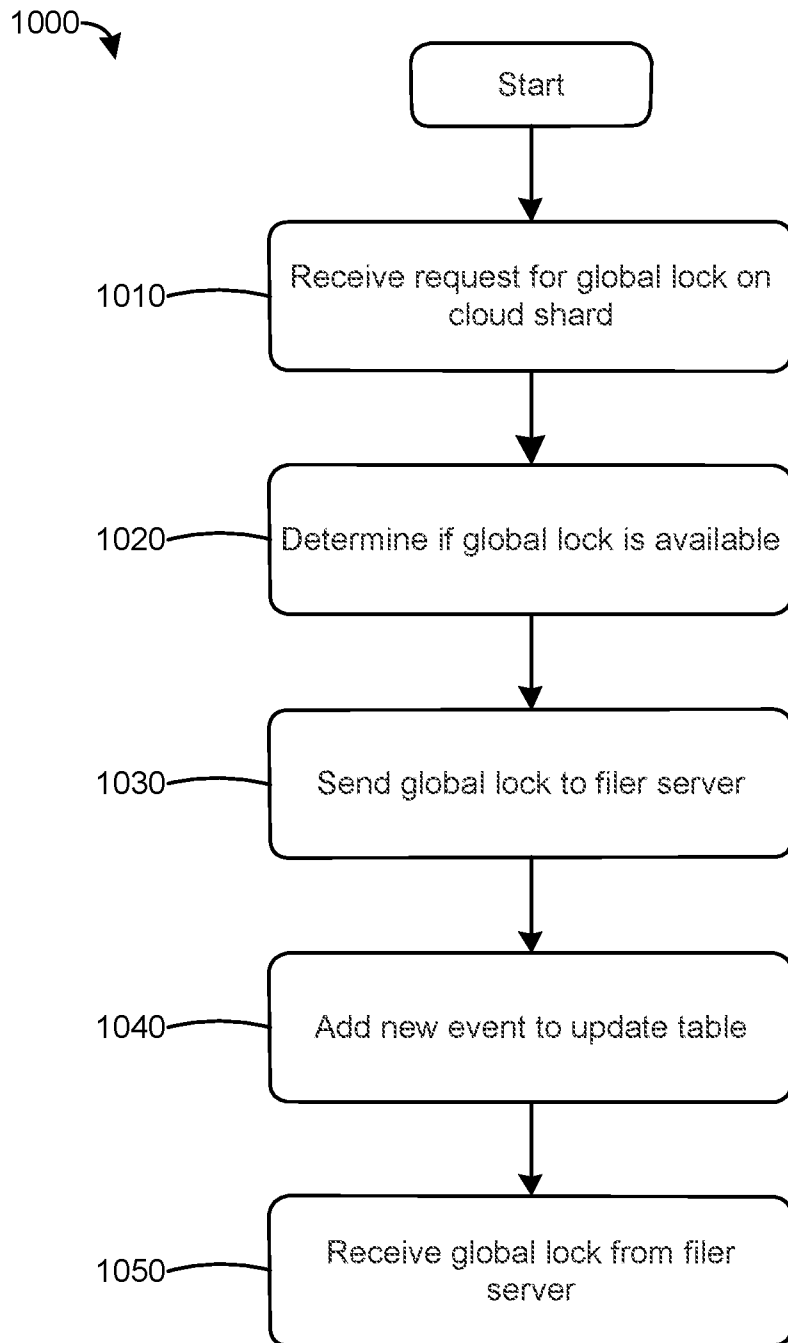


Fig. 10

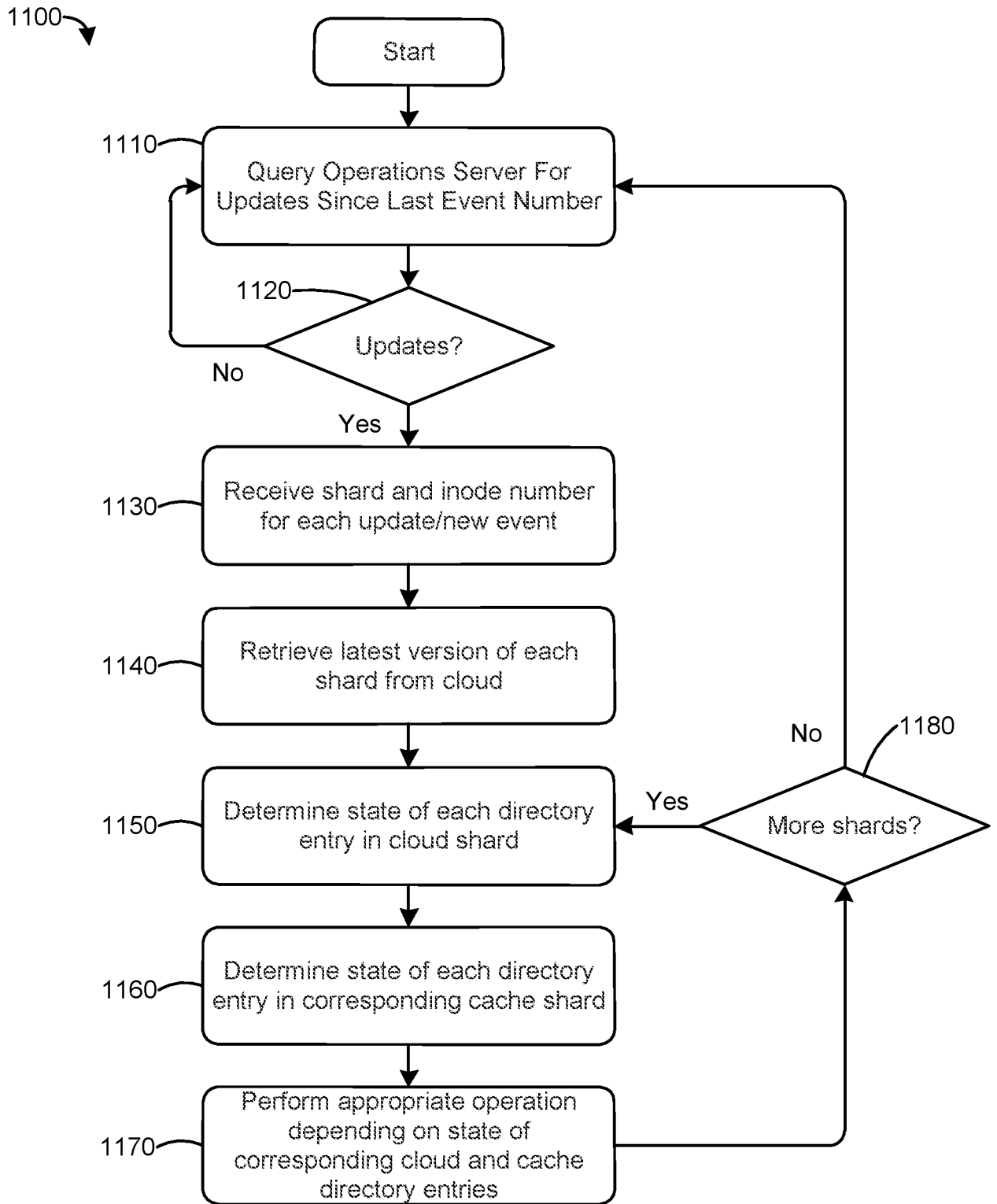


Fig. 11


1200 →

```
/inodes/1/S1/now ("latest" = "1")
```

```
/inodes/1/S1/1
```

```
<manifest>  
<inode>1</inode>  
<shard>1</shard>  
<contents>  
  <direntary>  
    <inode>100</inode>  
    <name>file1.txt</name>  
    <stbuf>  
      <nlink>1</nlink>  
      <size>1024</size>  
    </stbuf>  
  </direntary>  
</contents>  
</manifest>
```

Fig. 12

1300 

```
/inodes/1/S1/now ("latest" = "2")  
/inodes/1/S1/2
```

```
<manifest>  
<inode>1</inode>  
<shard>1</shard>  
<contents>  
  <dirent>  
    <inode>100</inode>  
    <name>file1.txt</name>  
    <stbuf>  
      <nlink>1</nlink>  
      <size>1024</size>  
    </stbuf>  
  </dirent>  
  <dirent>  
    <inode>101</inode>  
    <name>file2.txt</name>  
    <stbuf>  
      <nlink>1</nlink>  
      <size>2048</size>  
    </stbuf>  
  </dirent>  
</contents>  
</manifest>
```

Fig. 13

```
/inodes/101/now ("latest" = "1")  
/inodes/101/1  
1400 →  
<manifest>  
<inode>101</inode>  
<chunks>  
  <chunk  
    offset="0" length="1024"  
    handle="c1"/>  
  <chunk  
    offset="1024" length="1024"  
    handle="c2"/>  
</chunks>  
</manifest>
```

Fig. 14

1504
 /inodes/1/S1/now ("latest" = "2")
 /inodes/1/S1/2

<manifest>
 <inode>1</inode>
 <shard>1</shard>
 <contents>
 <dirententry>
 <inode>100</inode>
 <name>file1.txt</name>
 <stbuf>

<size>1024</size>
 </stbuf>
 </dirententry>
 <dirententry>
 <inode>101</inode>
 <name>file2.txt</name>
 <stbuf>

<size>2048</size>
 </stbuf>
 </dirententry>
 </contents>
 </manifest> 1500

1506
 /inodes/101/now ("latest" = "1")
 /inodes/101/1

<manifest>
 <inode>101</inode>
 <chunks>
 <chunk
 offset="0" length="1024"
 handle="c1"/>
 <chunk
 offset="1024" length="1024"
 handle="c2"/>
 </chunks>
 </manifest> 1502

1508
 /chunks/c1/data
 /chunks/c1/refs/100/1

/chunks/c2/data
 /chunks/c2/refs/100/1

FIG. 15

A. CLASSIFICATION OF SUBJECT MATTER**G06F 17/30(2006.01)i**

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHEDMinimum documentation searched (classification system followed by classification symbols)
G06F 17/30; H04L 29/08; H04L 12/24Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility modelsElectronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: cloud, server, update, file, shard, database, request, synchronize, compare, and similar terms.**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2014-0012814 A1 (BOX, INC.) 09 January 2014 See paragraphs [0016]-[0018], [0025], and [0050]-[0051]; claims 13 and 19; and figures 1 and 3.	1-17
A	US 2015-0229715 A1 (LINKEDIN CORPORATION) 13 August 2015 See paragraphs [0054] and [0061]-[0062]; and figures 4 and 6.	1-17
A	US 2014-0280276 A1 (TACTILE, INC.) 18 September 2014 See paragraphs [0026]-[0032] and figures 2-3.	1-17
A	US 2014-0237090 A1 (FACEBOOK, INC.) 21 August 2014 See paragraphs [0062]-[0063] and figure 4.	1-17
A	US 2014-0059001 A1 (LSIS CO., LTD.) 27 February 2014 See paragraphs [0032]-[0040] and figure 1.	1-17

 Further documents are listed in the continuation of Box C. See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

28 September 2017 (28.09.2017)

Date of mailing of the international search report

28 September 2017 (28.09.2017)

Name and mailing address of the ISA/KR

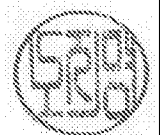
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea

Facsimile No. +82-42-481-8578

Authorized officer

NHO, Ji Myong

Telephone No. +82-42-481-8528



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/032004

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2014-0012814 A1	09/01/2014	None	
US 2015-0229715 A1	13/08/2015	None	
US 2014-0280276 A1	18/09/2014	US 9424330 B2	23/08/2016
US 2014-0237090 A1	21/08/2014	US 9590852 B2	07/03/2017
US 2014-0059001 A1	27/02/2014	CN 103634369 A	12/03/2014
		EP 2701085 A1	26/02/2014
		JP 2014-041612 A	06/03/2014
		JP 5788934 B2	07/10/2015
		KR 10-1638656 B1	11/07/2016
		KR 10-2014-0025863 A	05/03/2014
		US 9477735 B2	25/10/2016