



US008805968B2

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

(10) **Patent No.:** US 8,805,968 B2
(45) **Date of Patent:** *Aug. 12, 2014

(54) **ACCESSING CACHED DATA FROM A PEER CLOUD CONTROLLER IN A DISTRIBUTED FILESYSTEM**

- (71) Applicant: **Panzura, Inc.**, Campbell, CA (US)
- (72) Inventors: **John Richard Taylor**, Tiburon, CA (US); **Randy Yen-pang Chou**, San Jose, CA (US); **Andrew P. Davis**, Santa Cruz, CA (US)
- (73) Assignee: **Panzura, Inc.**, Campbell, CA (US)
- (*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: 13/725,767

(22) Filed: Dec. 21, 2012

(65) **Prior Publication Data**

US 2013/0117240 A1 May 9, 2013

Related U.S. Application Data

- (63) Continuation-in-part of application No. 13/295,844, filed on Nov. 14, 2011, and a continuation-in-part of application No. 13/225,194, filed on Sep. 2, 2011, now Pat. No. 8,356,016, and a continuation-in-part of application No. 12/772,927, filed on May 3, 2010, now Pat. No. 8,341,363.

(51) **Int. Cl.**
G06F 15/16 (2006.01)

(52) **U.S. Cl.**
USPC 709/219

(58) **Field of Classification Search**
USPC 709/219
See application file for complete search history.

(56)

References Cited

U.S. PATENT DOCUMENTS

6,466,978 B1	10/2002	Mukherjee et al.
6,697,846 B1	2/2004	Soltis
7,610,285 B1	10/2009	Zoellner et al.
7,624,134 B2	11/2009	Stakutis et al.
7,657,550 B2	2/2010	Prahlad et al.
7,844,582 B1	11/2010	Arbilla et al.
7,865,873 B1	1/2011	Zoellner et al.
8,296,398 B2 *	10/2012	Lacapra et al. 709/219
2007/0088702 A1 *	4/2007	Fridella et al. 707/10
2008/0162608 A1 *	7/2008	Torii et al. 707/204
2010/0100698 A1	4/2010	Yang et al.
2010/0114832 A1	5/2010	Lillibridge et al.
2010/0325377 A1	12/2010	Lango et al.
2010/0333116 A1 *	12/2010	Prahlad et al. 719/328
2011/0066668 A1 *	3/2011	Guarraci 707/831
2011/0276713 A1	11/2011	Brand
2012/0011176 A1	1/2012	Aizman
2012/0089781 A1 *	4/2012	Ranade et al. 711/118
2013/0204849 A1 *	8/2013	Chacko 707/692

* cited by examiner

Primary Examiner — Kevin Bates

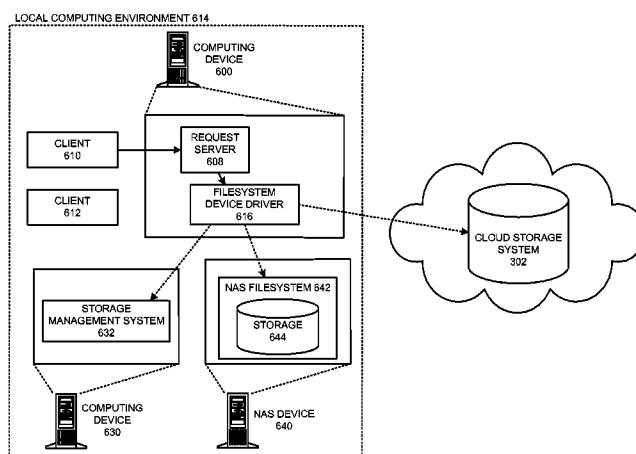
Assistant Examiner — S M Rahman

(74) *Attorney, Agent, or Firm* — Park, Vaughan, Fleming & Dowler LLP; Mark Spiller

(57) **ABSTRACT**

The disclosed embodiments provide a system that archives data for a distributed filesystem. Two or more cloud controllers collectively manage distributed filesystem data that is stored in one or more cloud storage systems; the cloud controllers cache and ensure data consistency for the stored data. During operation, a cloud controller receives a request from a client for a data block of a file stored in the distributed filesystem. Upon determining that the requested data block is not currently cached in the cloud controller, the cloud controller sends a peer cache request for the requested data block to a peer cloud controller in the distributed filesystem.

20 Claims, 25 Drawing Sheets



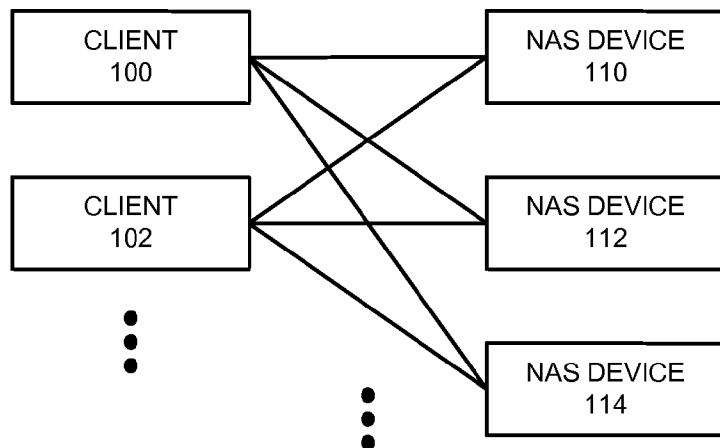


FIG. 1A

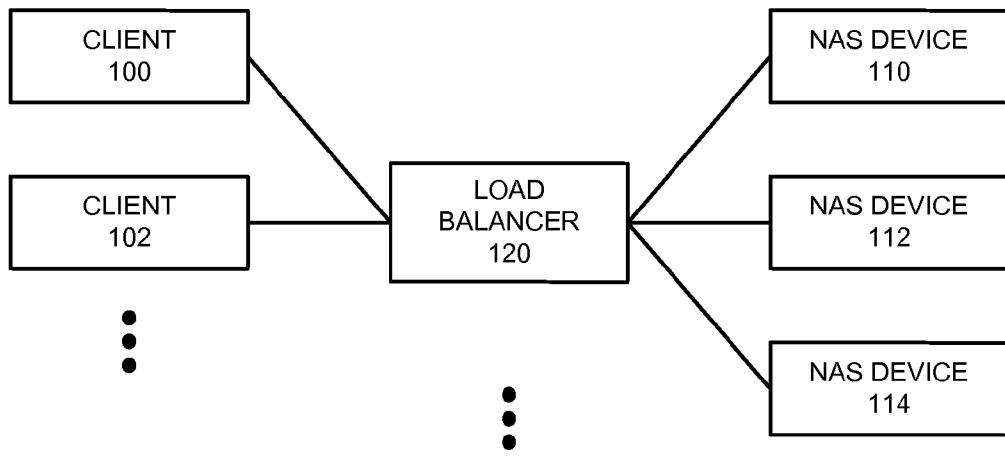


FIG. 1B

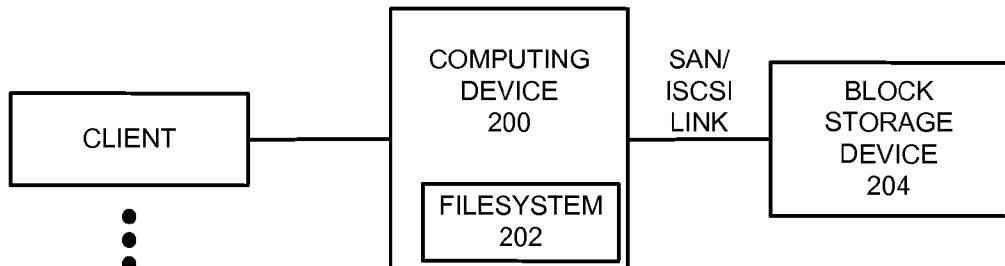
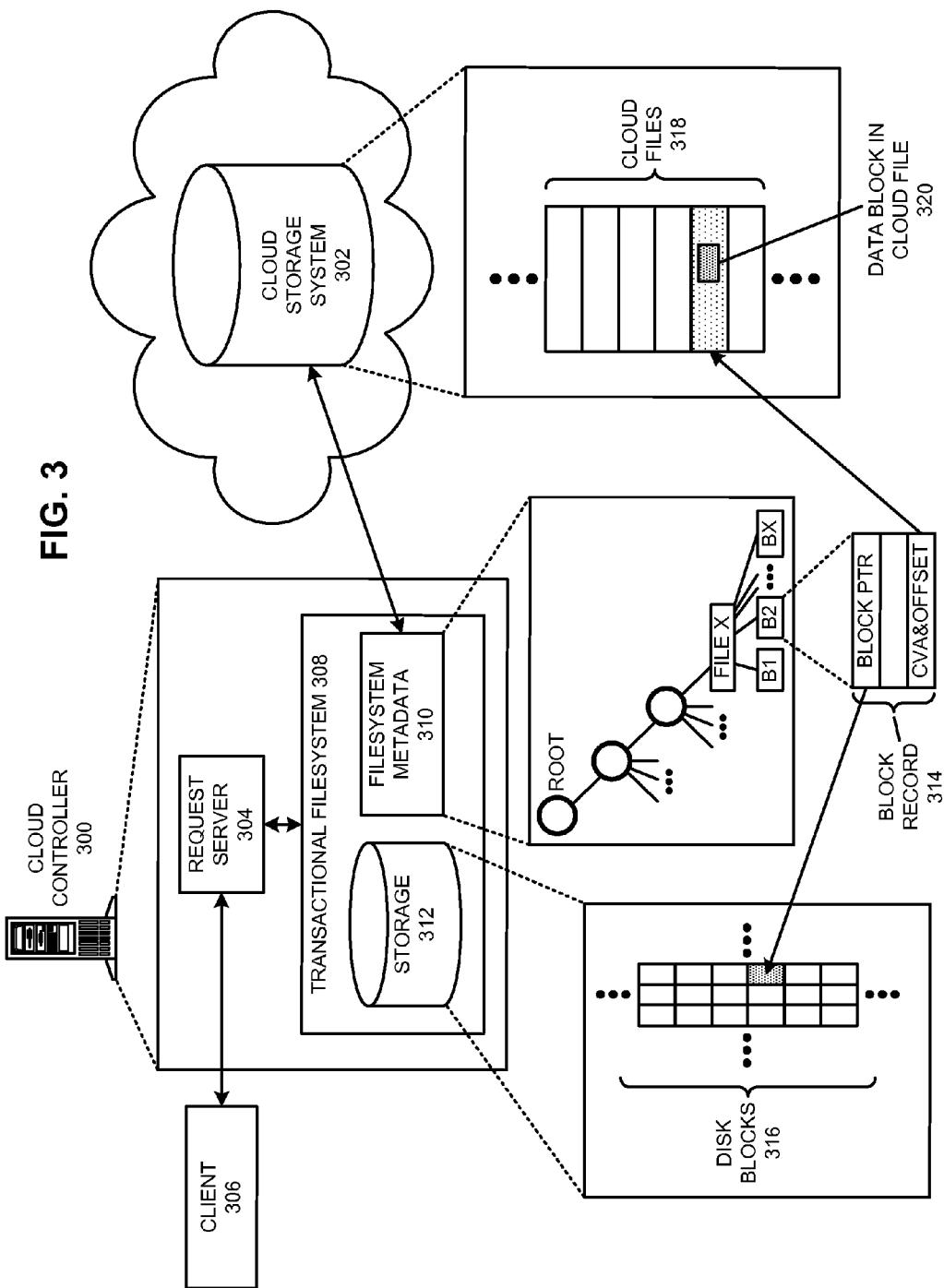
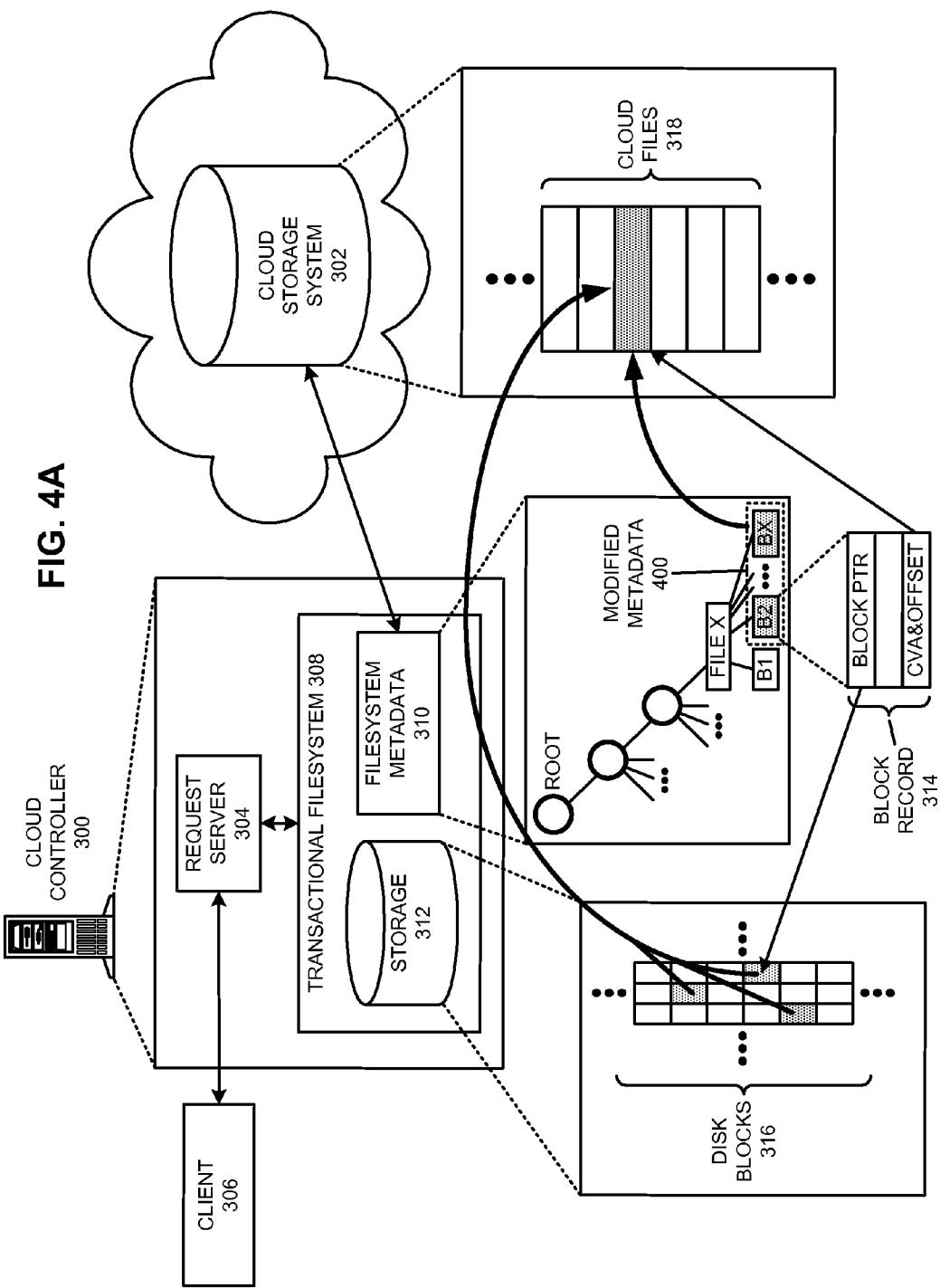


FIG. 2





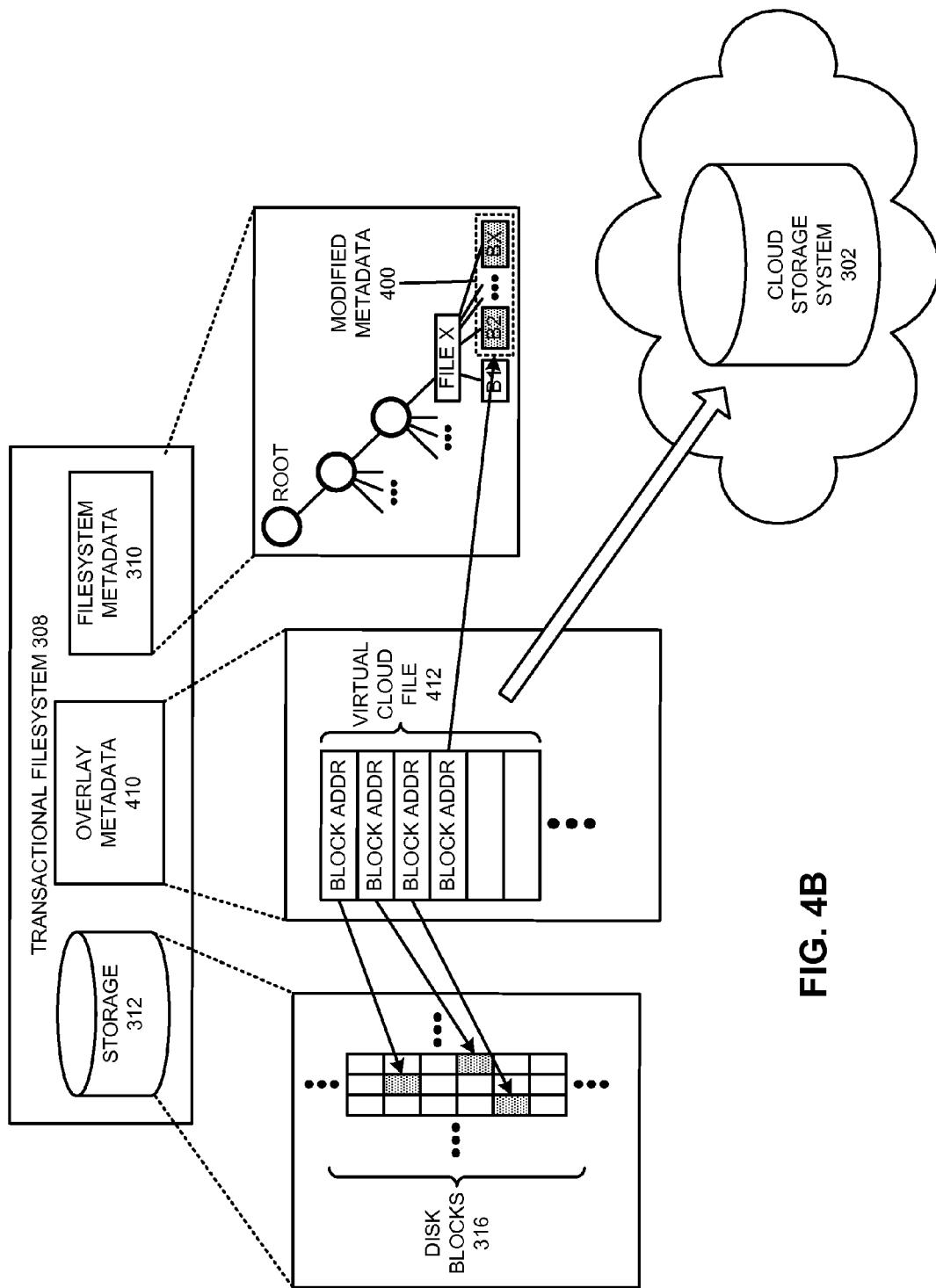


FIG. 4B

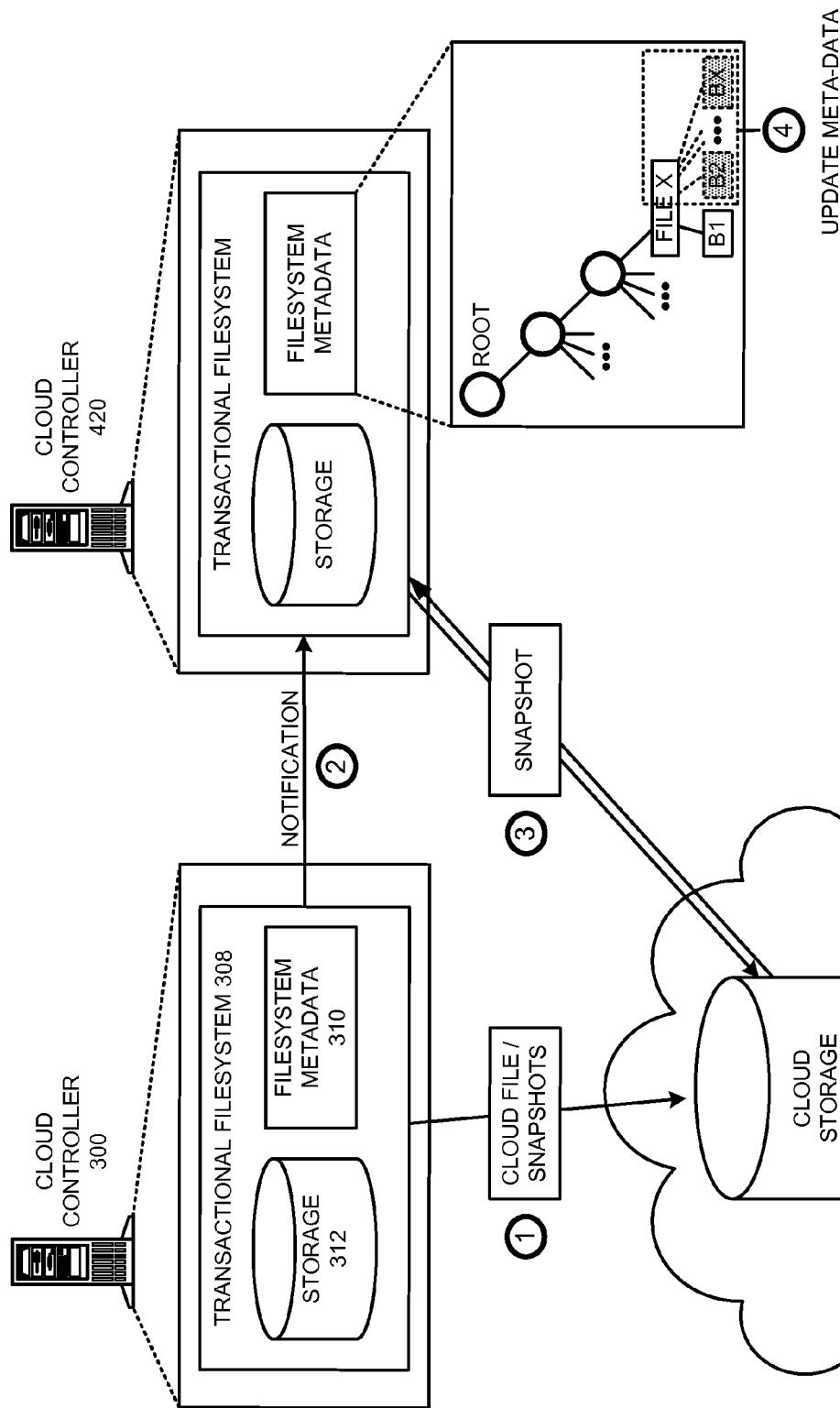
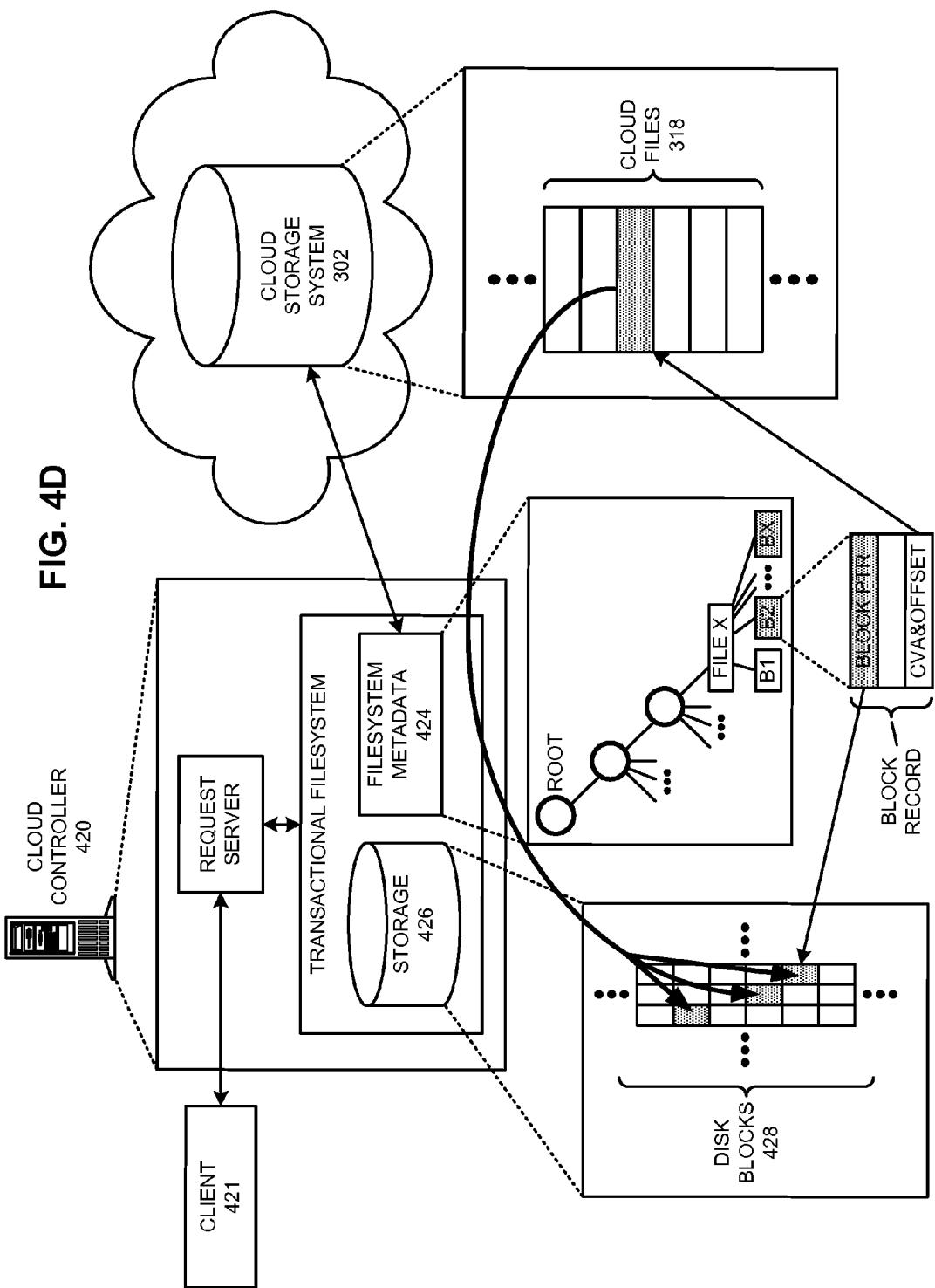


FIG. 4C



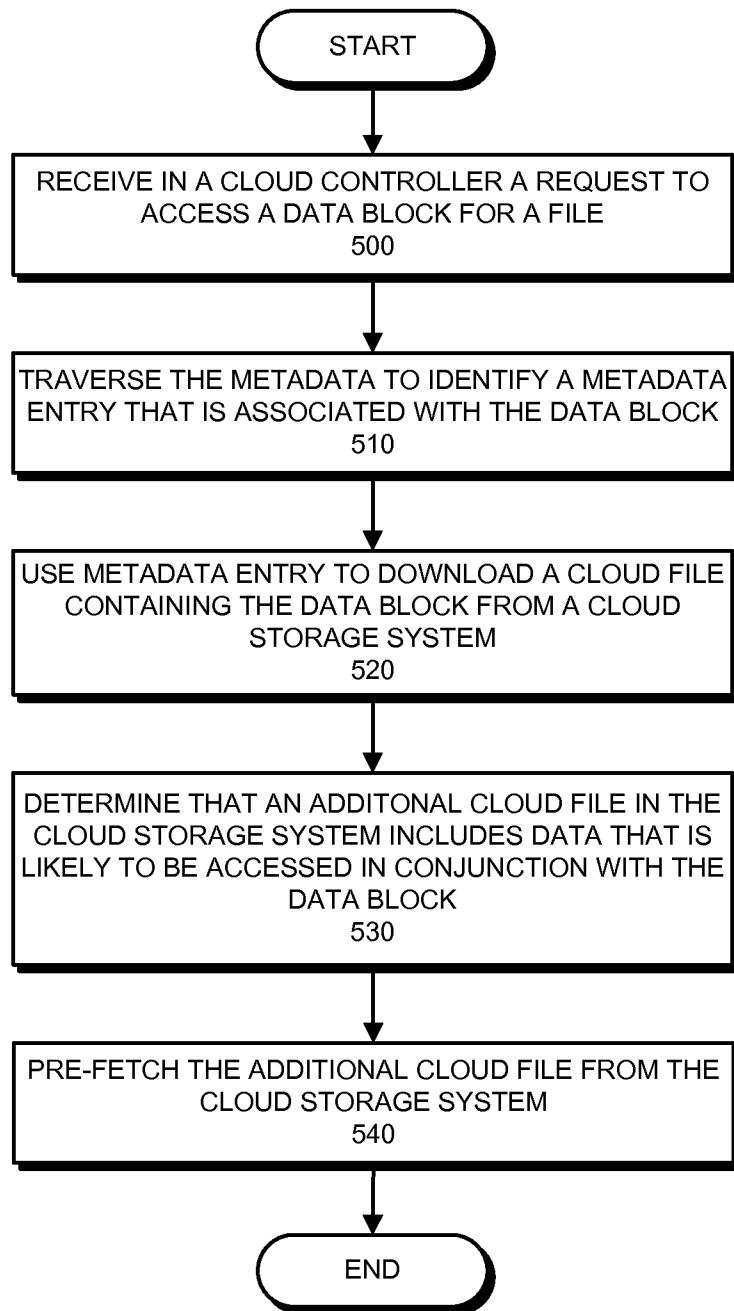
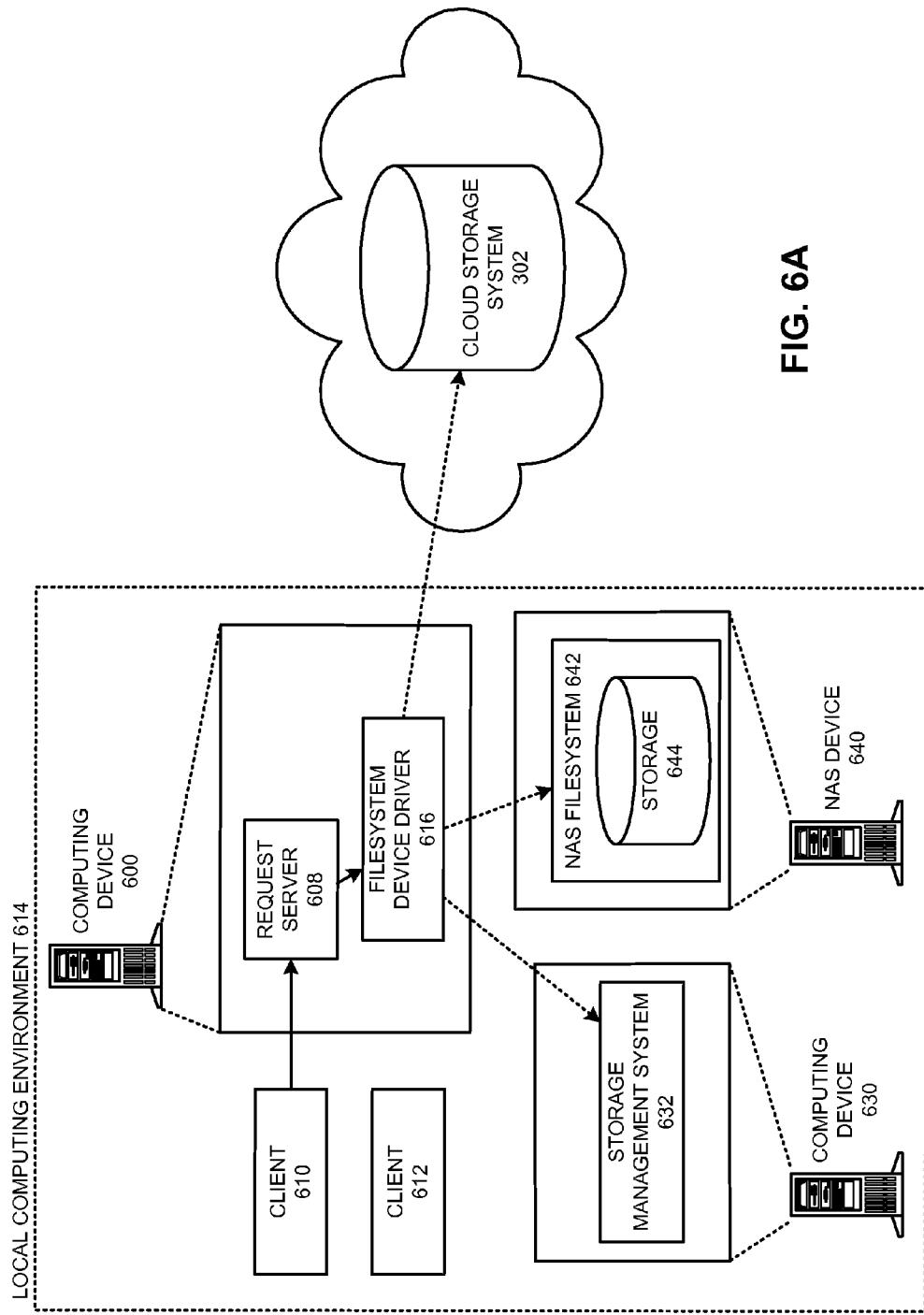
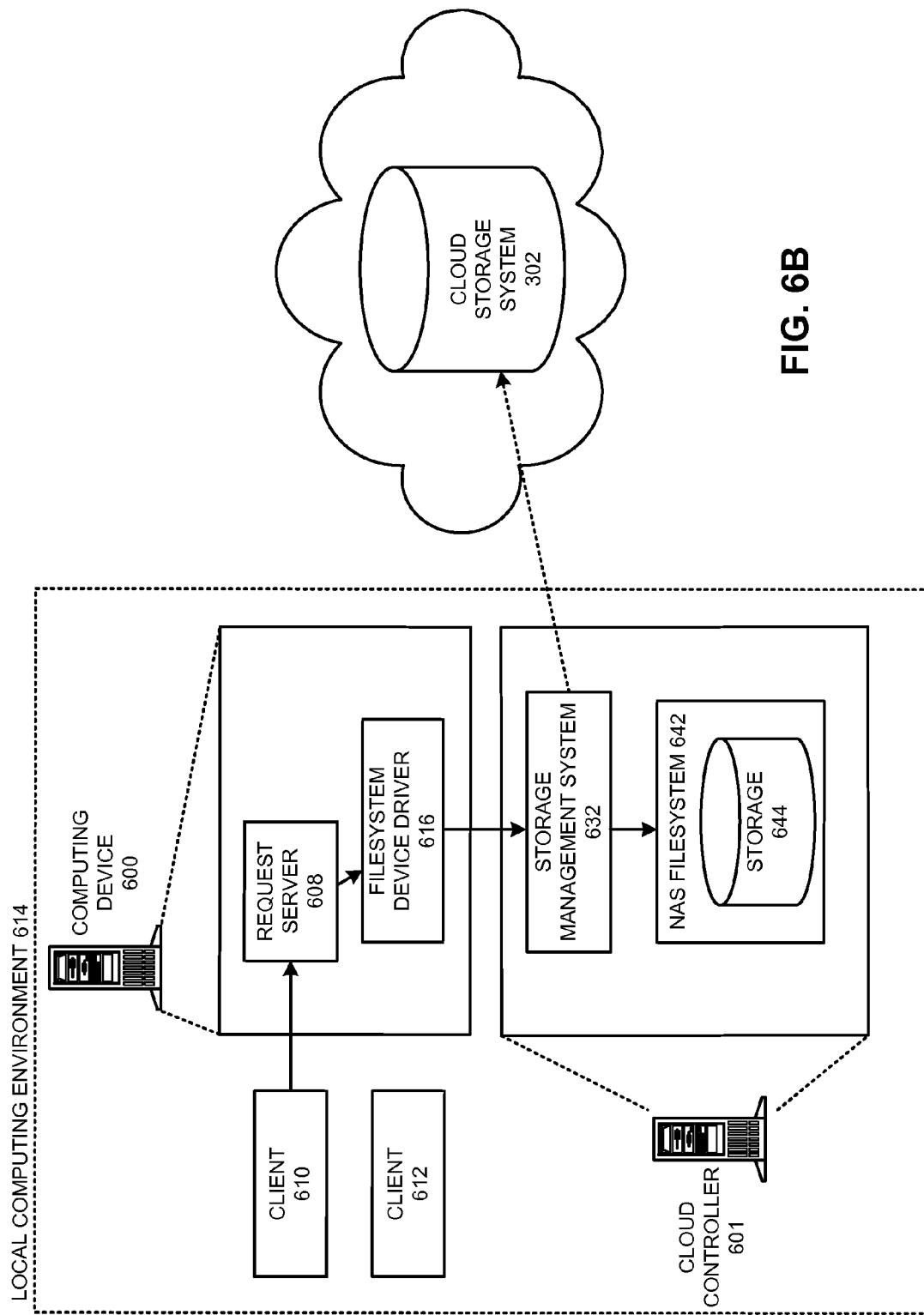
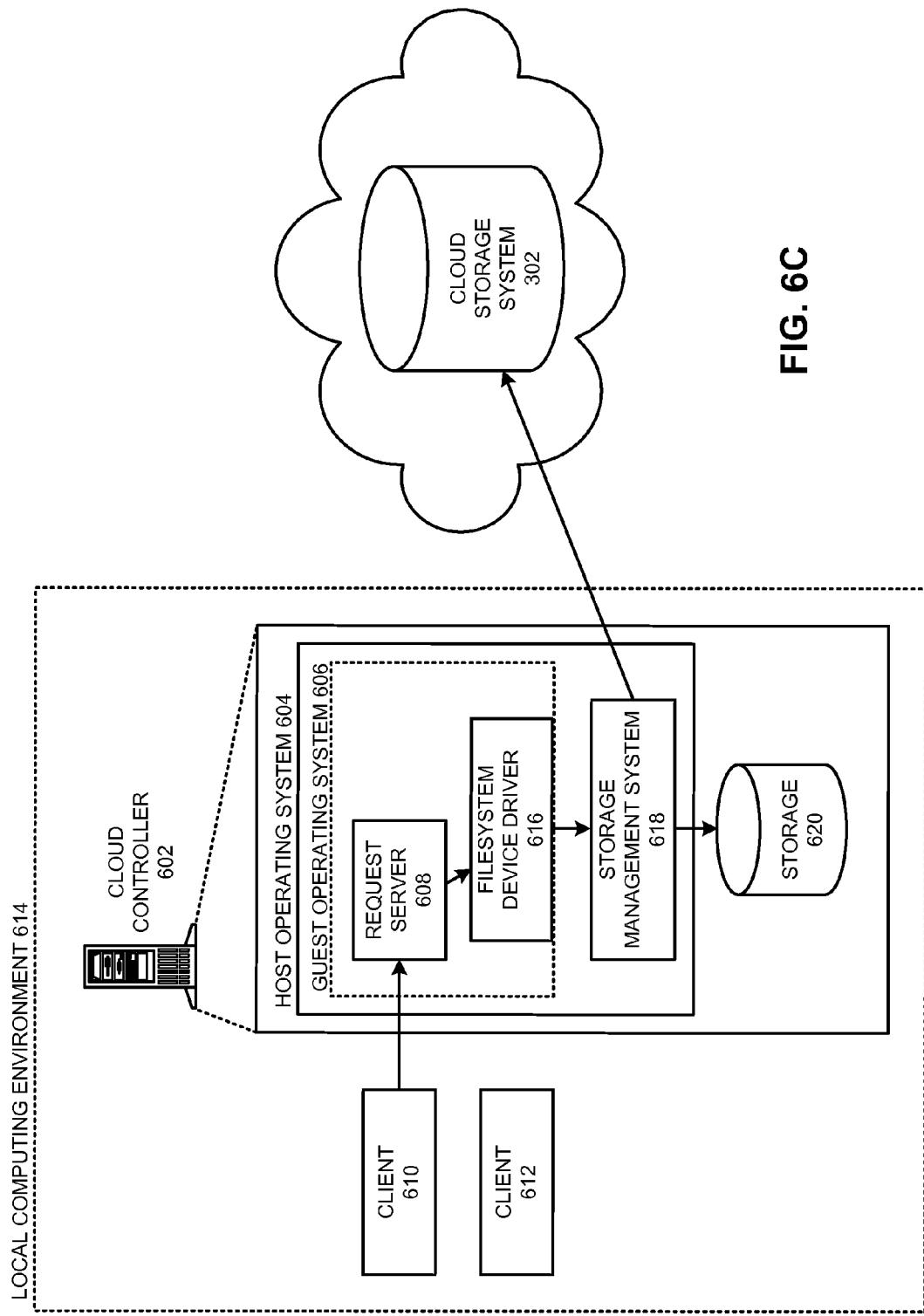
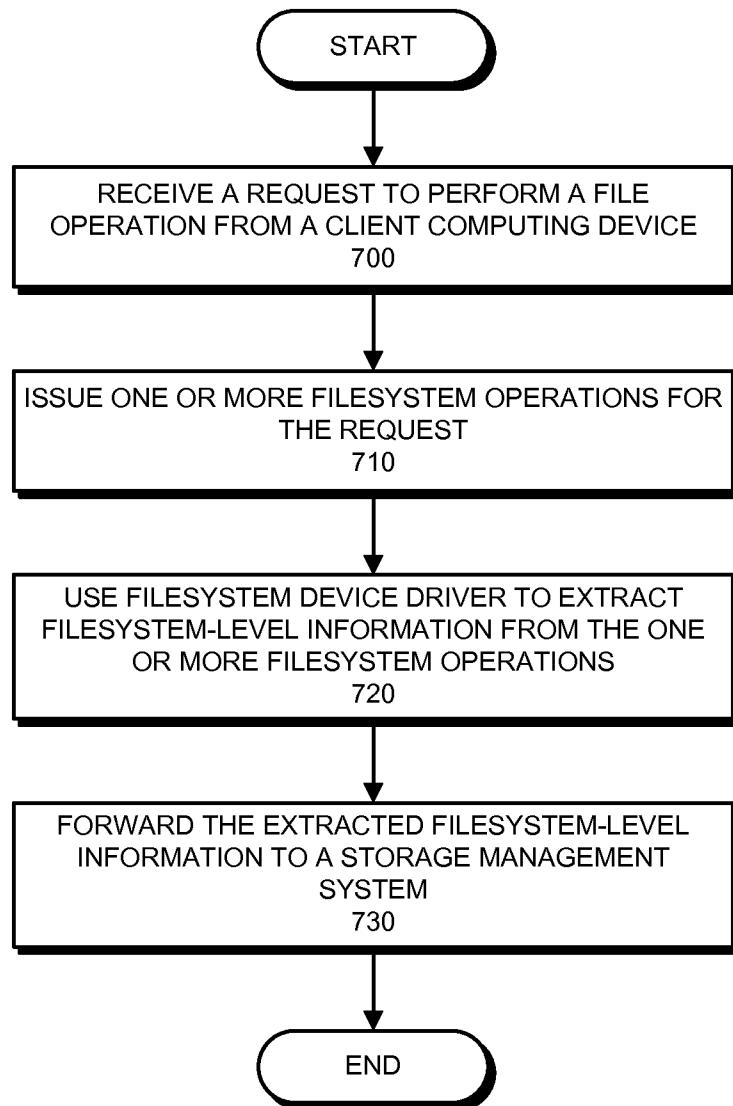


FIG. 5







**FIG. 7A**

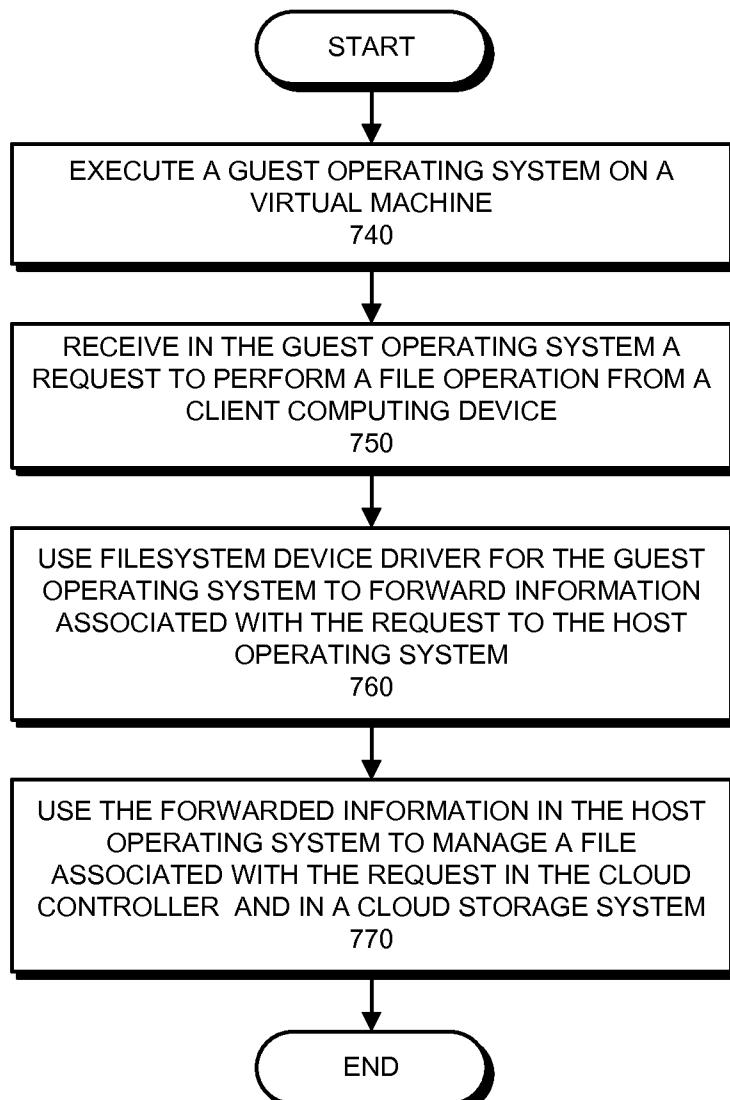
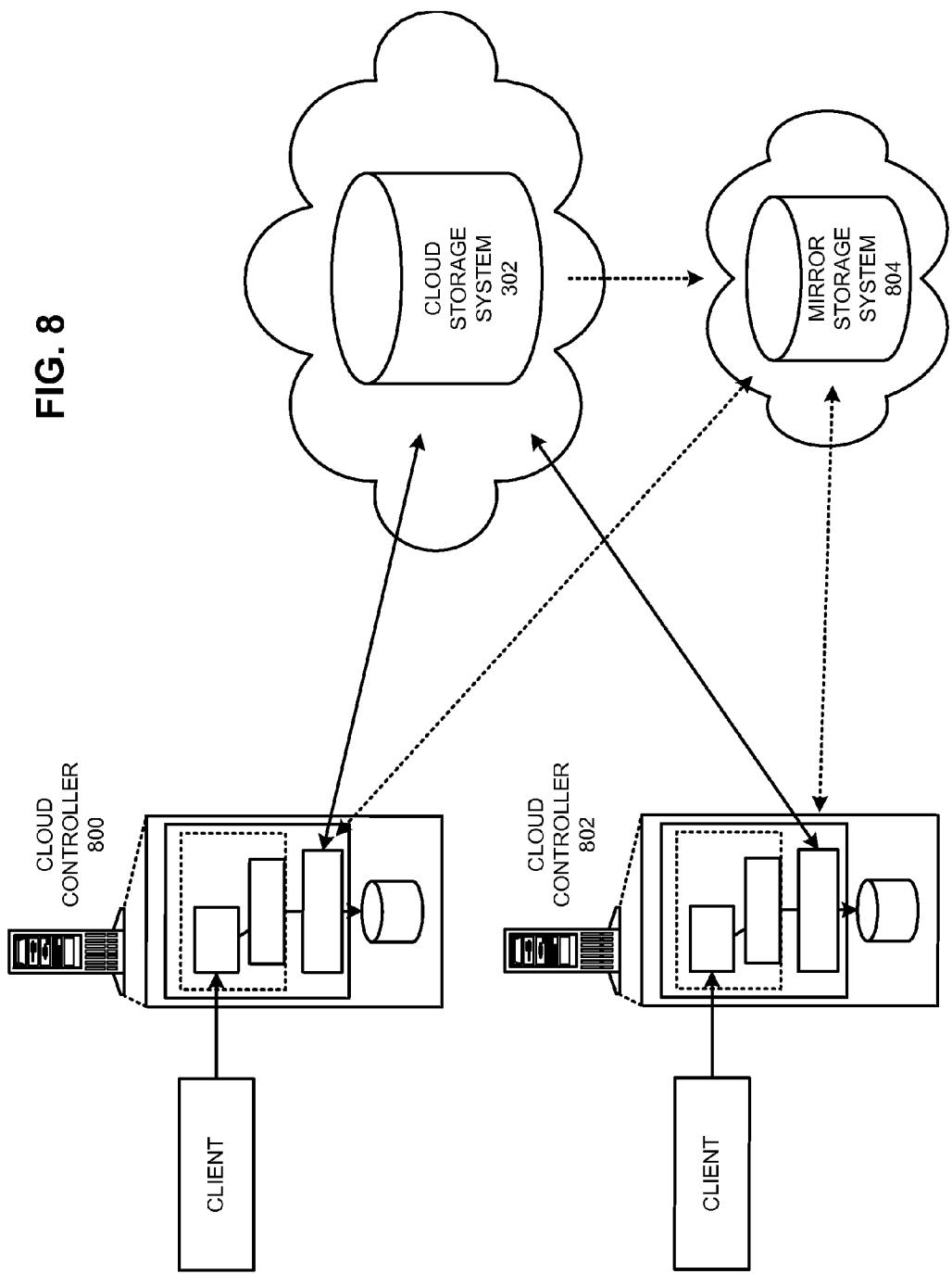
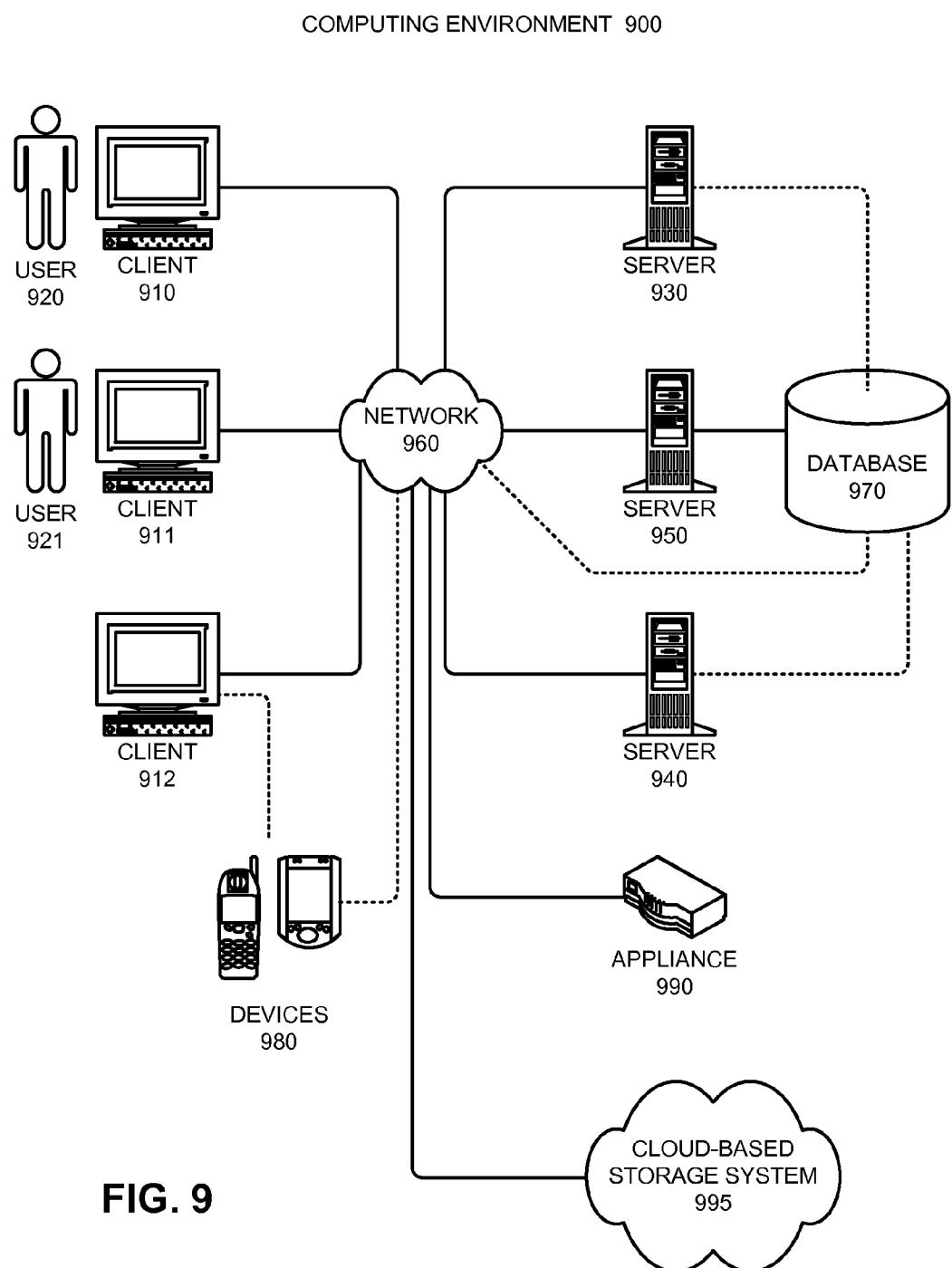
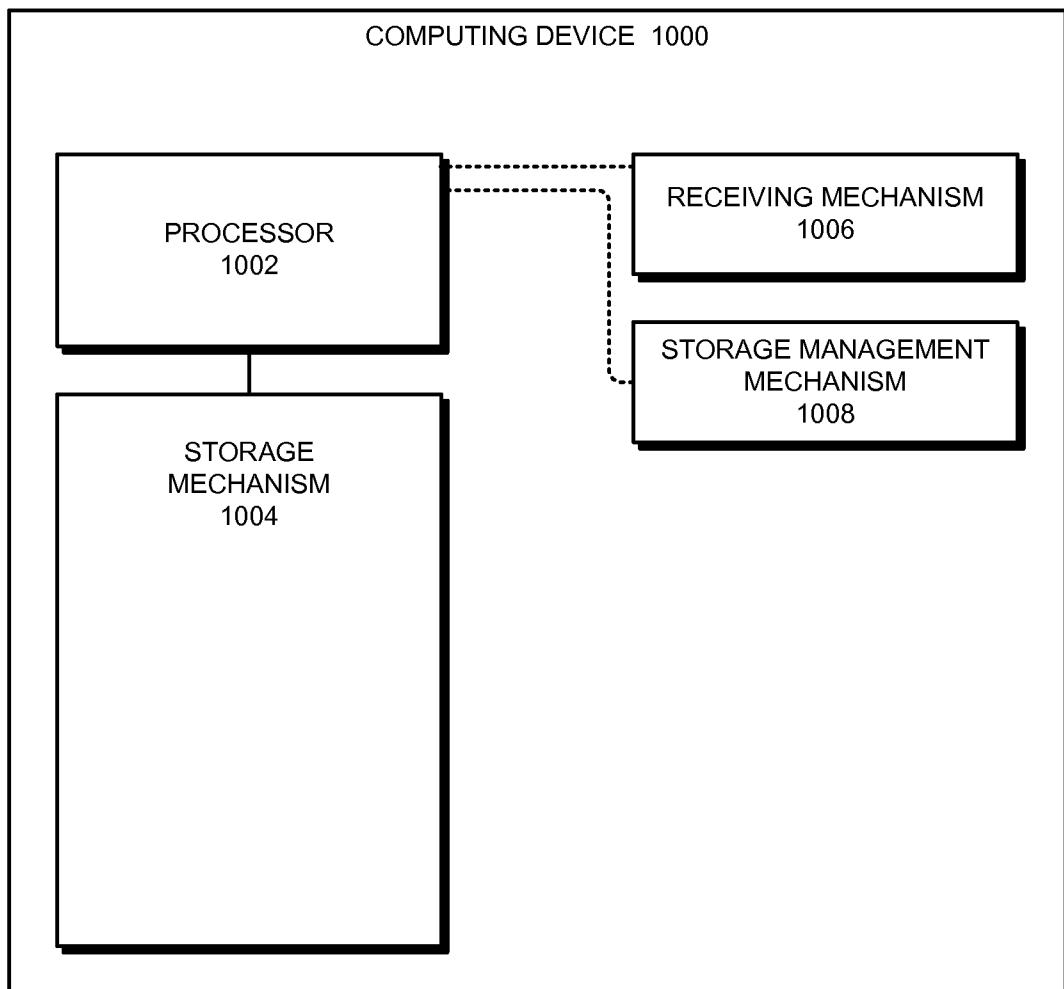
**FIG. 7B**

FIG. 8

**FIG. 9**

**FIG. 10**

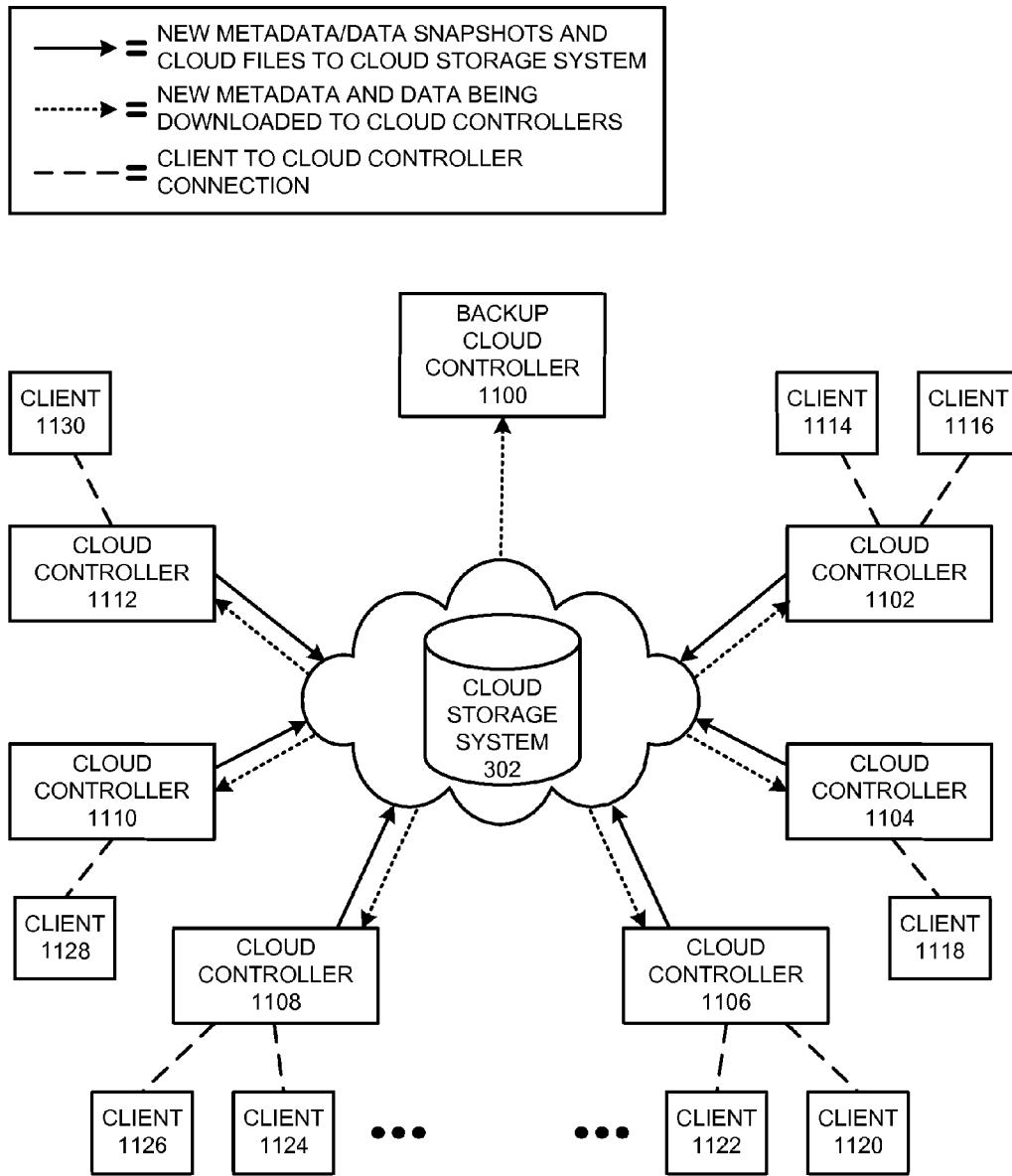


FIG. 11A

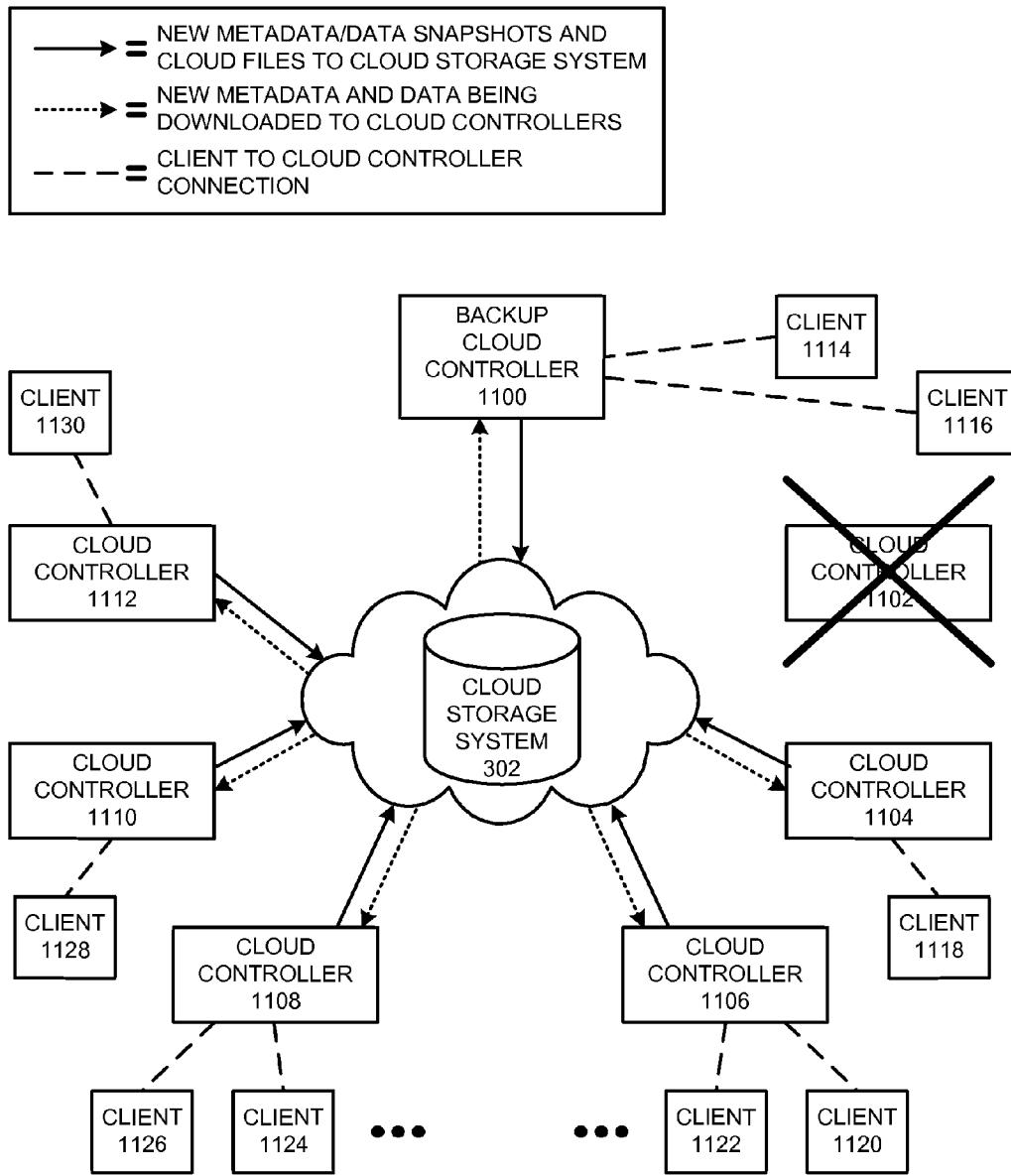
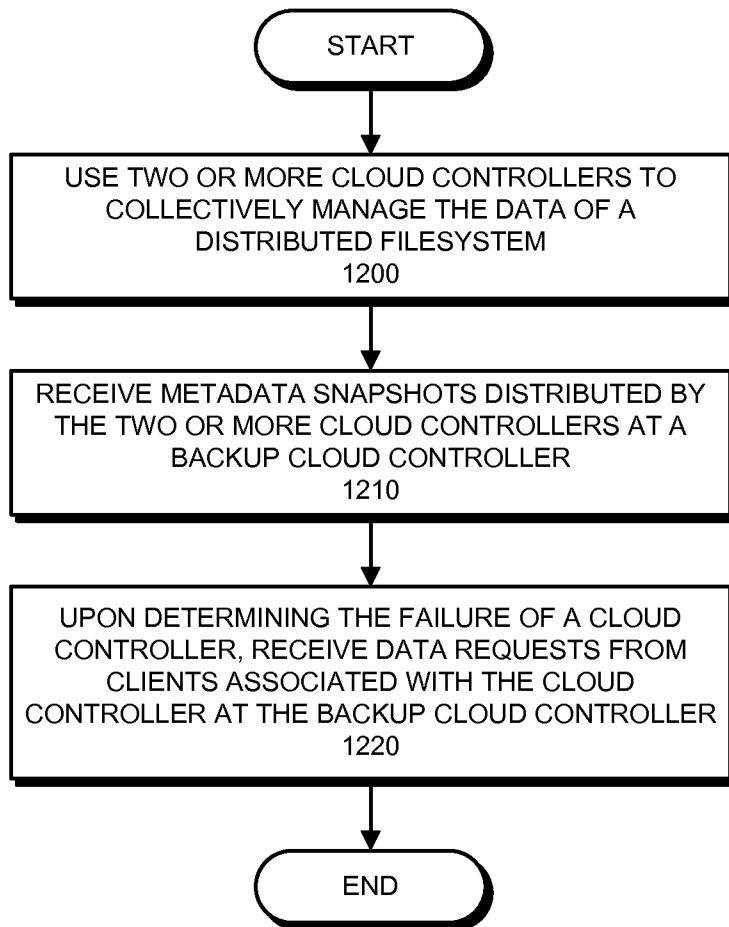
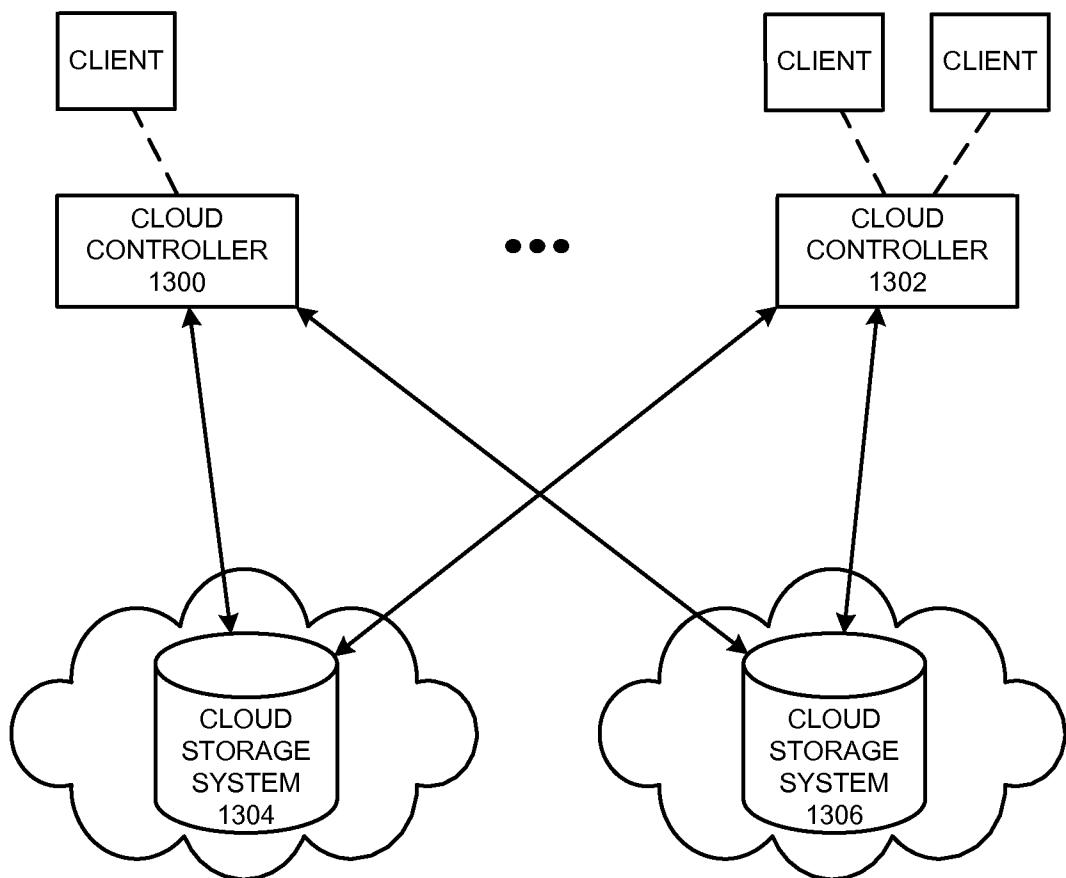
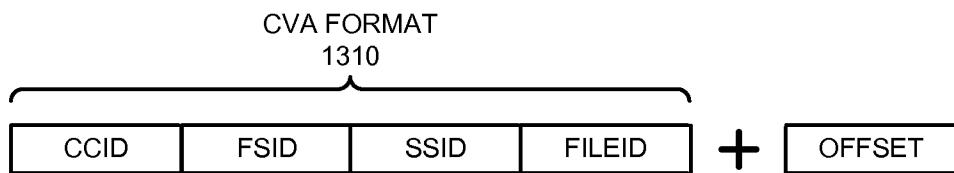
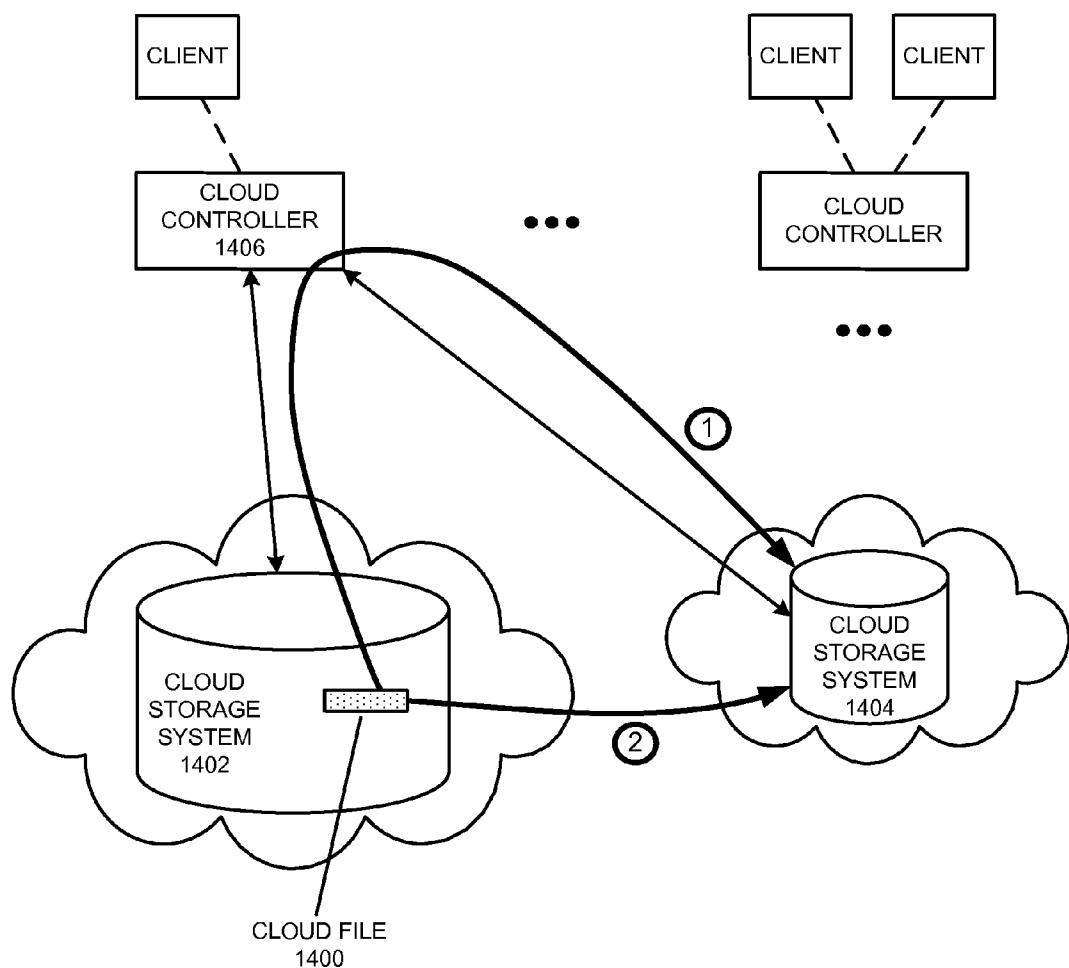
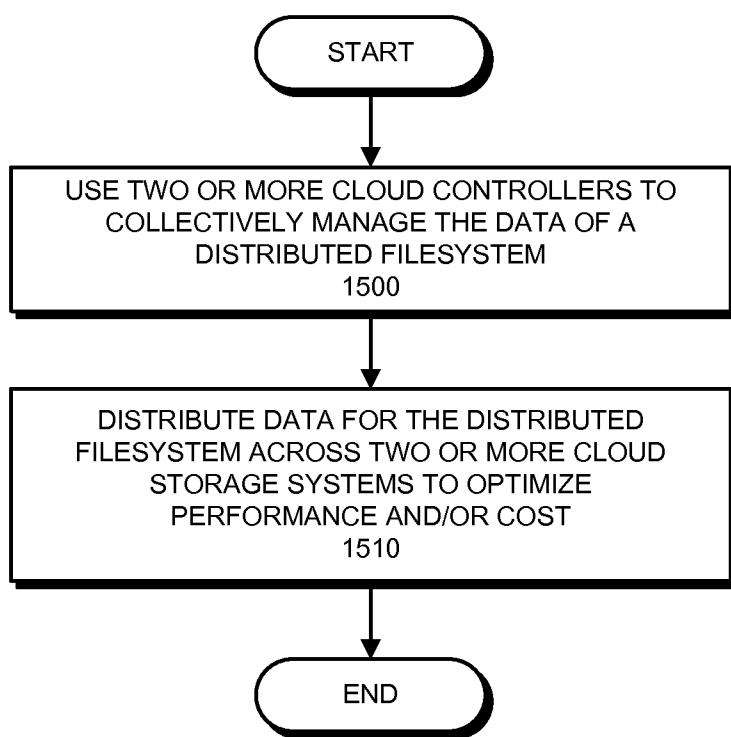


FIG. 11B

**FIG. 12**

**FIG. 13A****FIG. 13B**

**FIG. 14**

**FIG. 15**

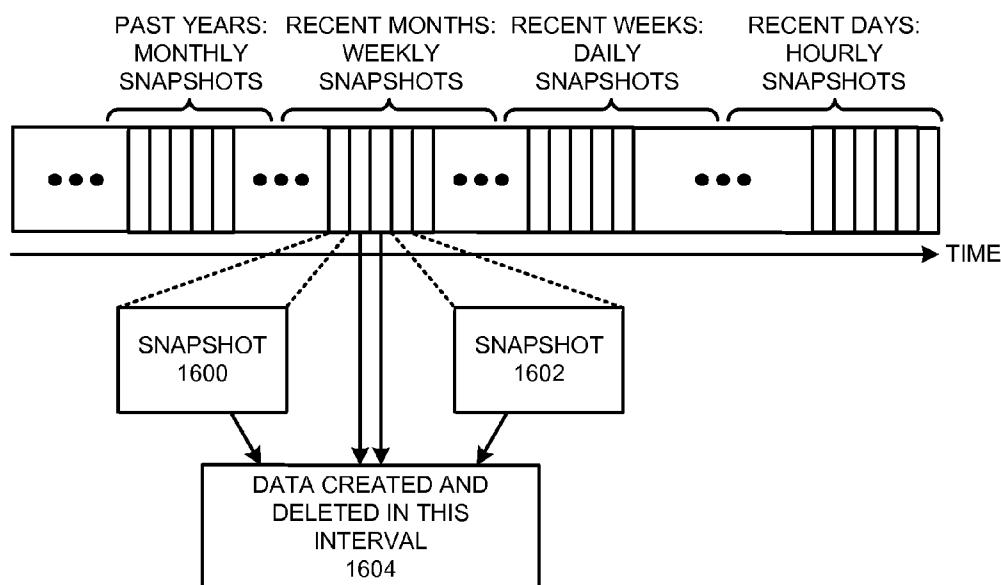


FIG. 16A

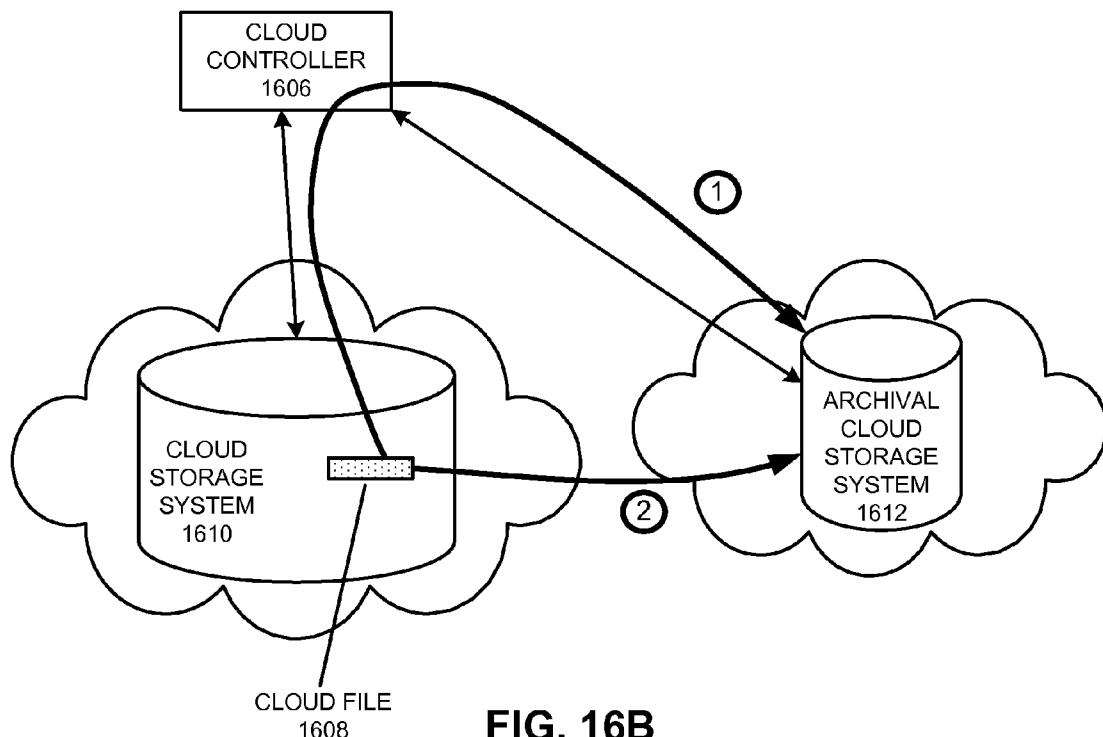
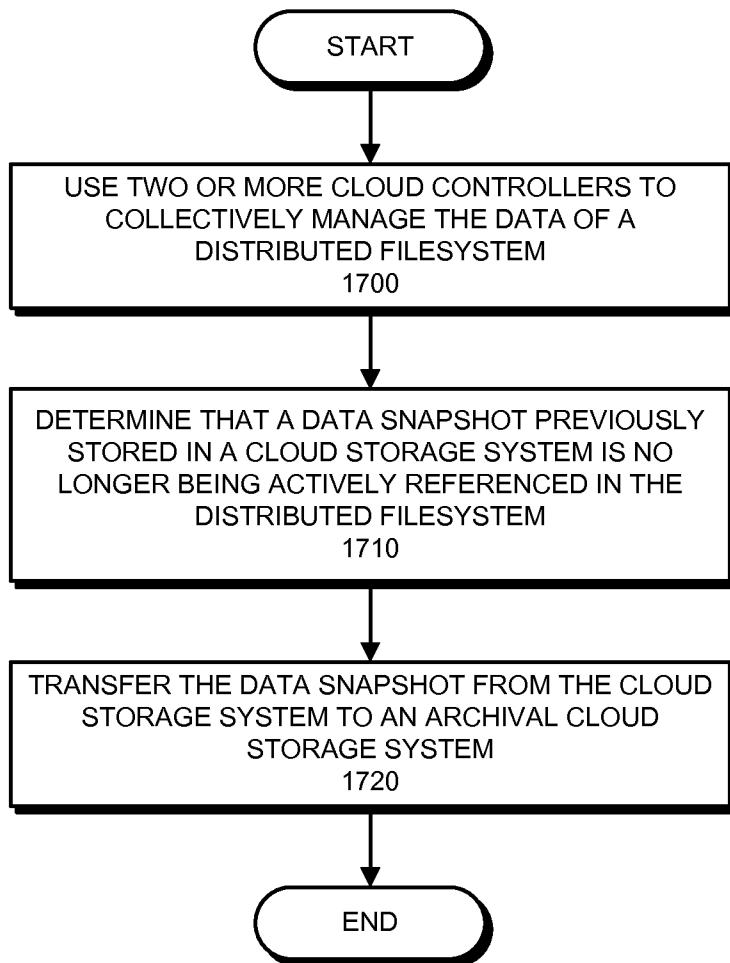


FIG. 16B

**FIG. 17**

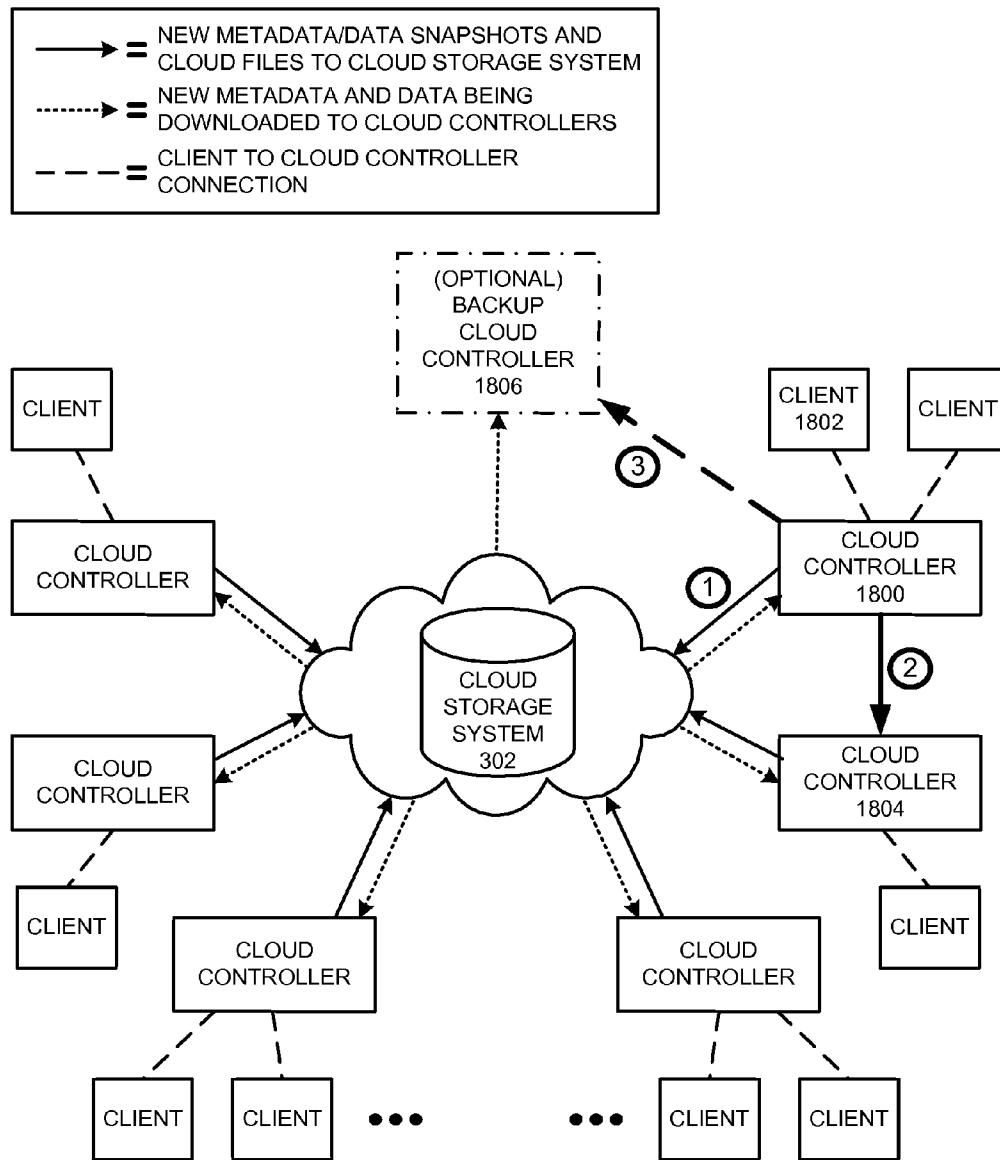
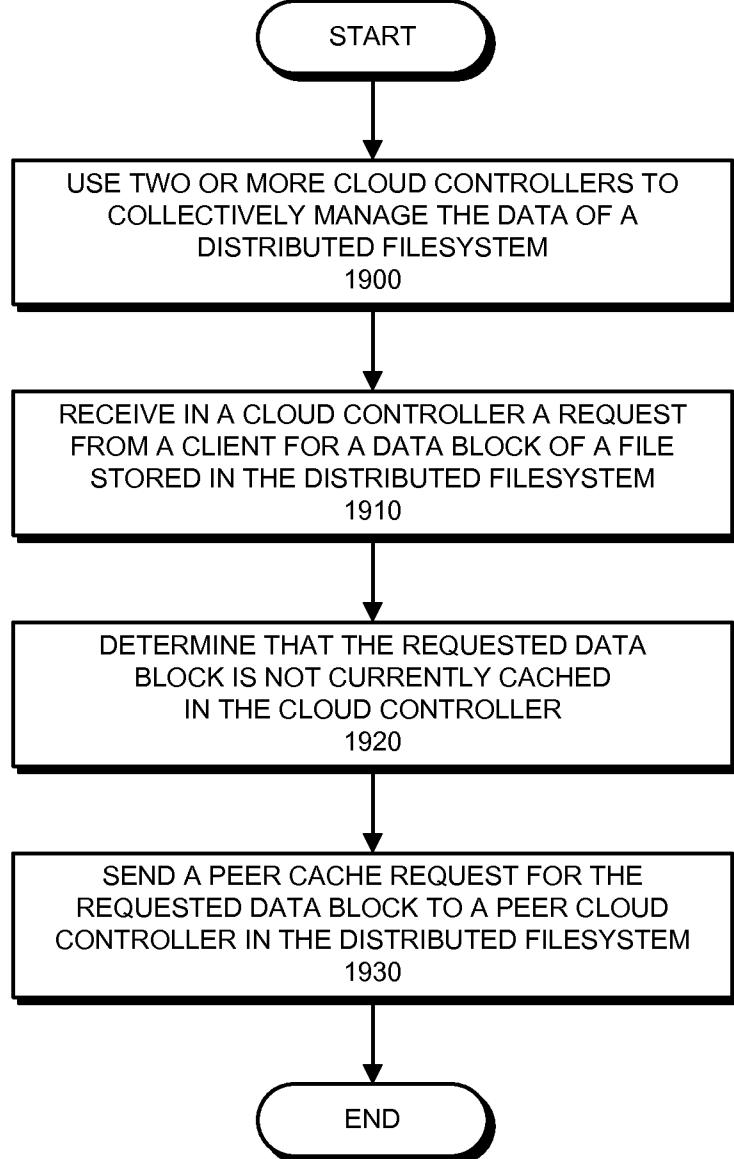


FIG. 18

**FIG. 19**

ACCESSING CACHED DATA FROM A PEER CLOUD CONTROLLER IN A DISTRIBUTED FILESYSTEM

RELATED APPLICATIONS

This application is a continuation-in-part application of each of the following pending U.S. patent applications: (1) Ser. No. 12/772,927 filed on 3 May 2010 by inventors Randy Yen-pang Chou, Ravi Mulum, and Steve Jung, entitled, “Efficient Cloud Network Attached Storage”; (2) Ser. No. 13/225,194 filed on 2 Sep. 2011 by inventor Randy Yen-pang Chou, entitled “Forwarding Filesystem-Level Information to a Storage Management System;” and (3) Ser. No. 13/295,844 filed on 14 Nov. 2011 by inventors John Richard Taylor, Randy Yen-pang Chou, and Andrew P. Davis, entitled “Pre-Fetching Data for a Distributed Filesystem.” This application hereby claims priority under 35 U.S.C. §120 to the above-listed pending U.S. patent applications. The contents of U.S. patent application Ser. Nos. 12/772,927, 13/225,194, and 13/295,844 are incorporated by reference in their entirety.

BACKGROUND

1. Field of the Invention

This disclosure generally relates to techniques for providing flexible and extensible network storage systems. More specifically, this disclosure relates to techniques for storing and accessing data in a distributed filesystem.

2. Related Art

Enterprise data centers typically include large numbers of storage devices that can be accessed using high-speed networks. However, the management overhead for a large number of storage devices can become substantial. For instance, maintaining data consistency, redundancy, and storage system performance in the presence of hard drive failures can involve substantial time, effort, and expertise.

A number of “cloud-based storage” vendors attempt to simplify storage management by providing large-scale remote network storage solutions. Such vendors can leverage economies of scale to provide extensive data storage capacity that can be leased and accessed by clients. Clients can leverage such storage solutions to offload storage management overhead and to quickly and easily increase their data storage capacity on an as-needed basis. However, cloud-based storage involves another set of inherent risks and overheads. For instance, storing data remotely (“in the cloud”) often increases access latency, and multiple clients simultaneously accessing a shared data set in a cloud-based storage system may suffer from data consistency problems. Furthermore, network failures and/or outages in cloud-based storage systems can prevent clients from accessing their data for significant periods of time.

Hence, what is needed are techniques for providing network-based storage capabilities without the above-described problems of existing techniques.

SUMMARY

The disclosed embodiments provide a system that distributes data for a distributed filesystem across multiple cloud storage systems. Two or more cloud controllers collectively manage distributed filesystem data that is stored in one or more cloud storage systems; the cloud controllers cache and ensure data consistency for the stored data. Whenever each cloud controller receives new data from a client, the cloud controller generates a snapshot; determines an incremental

difference between the new snapshot and a previous snapshot; divides this incremental difference into an incremental metadata snapshot and an incremental data snapshot; and sends both the incremental metadata snapshot and the incremental data snapshot to the one or more cloud storage systems. The other controllers download the metadata, and optionally the data, from the cloud storage system. During operation, a cloud controller receives a request from a client for a data block of a file stored in the distributed filesystem.

Upon determining that the requested data block is not currently cached in the cloud controller, the cloud controller sends a peer cache request for the requested data block to a peer cloud controller in the distributed filesystem.

In some embodiments, a requesting cloud controller sends a peer cache request for a data block to a peer cloud controller in parallel with sending a request to the cloud storage system for a cloud file that contains requested data block. If the peer cloud controller is currently caching the data block, it sends the requested block to the requesting cloud controller; the requesting cloud controller can then cancel the request for the cloud file (or halt the transfer, if the cloud file is already being transferred). Alternatively, if the peer cloud controller is not currently caching the data block, the requesting cloud controller receives the requested cloud file from the cloud storage system, and extracts the requested data block from the received cloud file.

In some embodiments, the requesting cloud controller sends the peer cache request to the originating cloud controller (e.g., the cloud controller that originally uploaded the cloud file to the cloud storage system). Determining the originating cloud controller involves determining a cloud virtual address (CVA) for the cloud file in the cloud storage system that contains the data block and using a unique cloud controller identifier in this CVA to identify the originating cloud controller.

In some embodiments, a backup cloud controller for the distributed filesystem is configured to determine and cache a set of data that is most frequently accessed by the clients of the distributed filesystem. The requesting cloud controller determines the presence of this backup cloud controller, and sends a peer cache request for a desired data block to the backup cloud controller.

In some embodiments, two or more cloud controllers for the distributed filesystem may be co-located and configured to collaboratively cache data for a set of clients by tracking the set of data that is being cached by each co-located cloud controller. In such embodiments, one of these co-located cloud controllers may be configured to send peer cache requests to the other co-located cloud controller(s).

In some embodiments, a requesting cloud controller may determine one or more target peer cloud controllers based on a locality policy that specifies peer cloud controllers for the requesting cloud controller.

In some embodiments, determining a target peer cloud controller may involve determining the network latency and network bandwidth between the cloud controller and the other cloud controllers of the distributed filesystem, and sending a peer cache request to one or more cloud controllers that are in close network proximity to the requesting cloud controller.

In some embodiments, determining a target peer cloud controller may involve determining the load of one or more other cloud controllers of the distributed filesystem, and only sending peer cache requests to cloud controllers that have a low load.

In some embodiments, a cloud controller is configured to cache data for an associated set of clients during a time

interval in which the clients are active, but then caches data for another (requesting) cloud controller during a second time interval in which its local clients are idle. The requesting cloud controller is configured to send peer cache requests to this peer cloud controller during this second time interval.

In some embodiments, the cloud storage system is unaware of the organization and structure of the distributed filesystem. Data is stored in the distributed filesystem as cloud files that are uniquely indexed using a global address space. In this context, a second cloud controller receiving the incremental metadata snapshot sent by a first cloud controller can use a metadata entry in the incremental metadata snapshot to retrieve data from the cloud storage system. More specifically, the second cloud controller can: determine from the metadata entry that a desired data block in the incremental data snapshot is not currently cached in the second cloud controller; use a global address stored in the metadata entry to identify a cloud file in the cloud storage system that includes the data block; download the cloud file; and use an offset stored in the metadata entry to determine the location of the data block in the cloud file.

BRIEF DESCRIPTION OF THE FIGURES

FIG. 1A illustrates a set of clients that are configured to access NAS devices.

FIG. 1B illustrates a set of clients that are configured to access NAS devices via a load balancer.

FIG. 2 illustrates a network storage system that provides remote storage with a disk-level abstraction.

FIG. 3 illustrates an exemplary system in which a cloud controller manages and accesses data stored in a cloud storage system in accordance with an embodiment.

FIG. 4A illustrates the process of generating a cloud file for a snapshot in the context of the exemplary system of FIG. 3 in accordance with an embodiment.

FIG. 4B illustrates a set of overlay metadata and a virtual cloud file in the exemplary system of FIG. 3 in accordance with an embodiment.

FIG. 4C illustrates a second cloud controller that responds to a snapshot sent by the first cloud controller of FIGS. 3-4B in accordance with an embodiment.

FIG. 4D illustrates the process of accessing data from a cloud file in accordance with an embodiment.

FIG. 5 presents a flow chart that illustrates the process of pre-fetching data for a distributed filesystem in accordance with an embodiment.

FIG. 6A illustrates a computing device that receives and forwards requests for filesystem operations in accordance with an embodiment.

FIG. 6B illustrates a computing device that forwards requests for filesystem operations to a cloud controller in accordance with an embodiment.

FIG. 6C illustrates a cloud controller in which a filesystem device driver in a guest operating system forwards requests for filesystem operations in accordance with an embodiment.

FIG. 7A presents a flow chart that illustrates the process of forwarding filesystem-level information in accordance with an embodiment.

FIG. 7B presents a flow chart that illustrates the process of using a guest operating system to forward filesystem-level information in accordance with an embodiment.

FIG. 8 illustrates a distributed system in which updates are mirrored to an additional mirror storage system in accordance with an embodiment.

FIG. 9 illustrates a computing environment in accordance with an embodiment.

FIG. 10 illustrates a computing device in accordance with an embodiment.

FIG. 11A illustrates a distributed filesystem in which a backup cloud controller serves as a backup for set of cloud controllers that manage and access data stored in a cloud storage system in accordance with an embodiment.

FIG. 11B illustrates the distributed filesystem of FIG. 11A after the backup cloud controller has taken over the load of a failed cloud controller in accordance with an embodiment.

FIG. 12 presents a flow chart that illustrates the process of providing disaster recovery for a distributed filesystem in accordance with an embodiment.

FIG. 13A illustrates a distributed filesystem in which a set of cloud controllers manage and access data mirrored across multiple cloud storage systems in accordance with an embodiment.

FIG. 13B illustrates a cloud virtual address (CVA) format that facilitates identifying and locating cloud files across different cloud storage configurations in accordance with an embodiment.

FIG. 14 illustrates the process of migrating a cloud file from a first cloud storage system to a second cloud storage system in accordance with an embodiment.

FIG. 15 presents a flow chart that illustrates the process of distributing data for a distributed filesystem across multiple cloud storage systems in accordance with an embodiment.

FIG. 16A illustrates a set of snapshots that are taken for the distributed filesystem over time in accordance with an embodiment.

FIG. 16B illustrates the process of migrating a cloud file from a first cloud storage system to an archival cloud storage system in accordance with an embodiment.

FIG. 17 presents a flow chart that illustrates the process of archiving data for a distributed filesystem in accordance with an embodiment.

FIG. 18 illustrates a distributed filesystem in which a cloud controller issues peer cache requests to one or more peer cloud controllers in accordance with an embodiment.

FIG. 19 presents a flow chart that illustrates the process of archiving data for a distributed filesystem in accordance with an embodiment.

DETAILED DESCRIPTION

The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art, and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

The data structures and code described in this detailed description are typically stored on a non-transitory computer-readable storage medium, which may be any device or non-transitory medium that can store code and/or data for use by a computer system. The non-transitory computer-readable storage medium includes, but is not limited to, volatile memory, non-volatile memory, magnetic and optical storage devices such as disk drives, magnetic tape, CDs (compact discs), DVDs (digital versatile discs or digital video discs), or other media capable of storing code and/or data now known or later developed.

The methods and processes described in the detailed description section can be embodied as code and/or data, which can be stored in a non-transitory computer-readable storage medium as described above. When a computer system reads and executes the code and/or data stored on the non-transitory computer-readable storage medium, the computer system performs the methods and processes embodied as data structures and code and stored within the non-transitory computer-readable storage medium.

Furthermore, the methods and processes described below can be included in hardware modules. For example, the hardware modules can include, but are not limited to, application-specific integrated circuit (ASIC) chips, a full-custom implementation as part of an integrated circuit (or another type of hardware implementation on an integrated circuit), field-programmable gate arrays (FPGAs), a dedicated or shared processor that executes a particular software module or a piece of code at a particular time, and/or other programmable-logic devices now known or later developed. When the hardware modules are activated, the hardware modules perform the methods and processes included within the hardware modules.

Evolution of Network-Based Storage Systems

The proliferation of the Internet and large data sets have motivated a range of specialized data storage techniques. For instance, network-attached storage (NAS) devices often serve as centralized storage devices that provide large amounts of storage space for a set of heterogeneous clients in an enterprise. Such devices are typically tuned to provide a desired level of performance, redundancy (e.g., using a redundant array of independent disks (RAID)), and high availability. For example, while typical filesystems may take a substantial amount of time to recover from a crash (as the system has to process logs and/or journals to correctly rebuild modified data that was queued or in the process of being written at the time of the crash), NAS devices often incorporate transactional copy-on-write filesystems, which sacrifice some read performance in exchange for faster crash recovery. In a transactional copy-on-write filesystem, a file is not modified in place; instead, the system uses a delta encoding to append modifications (“deltas”) to the previous file data. Such encodings increase the overhead associated with read operations, because the system incurs additional computation and access time to read and process deltas stored at the end of a file. However, this encoding also ensures that files are “data-consistent” (e.g., reliably incorruptible and consistent across crashes and reboots), thereby allowing NAS devices to recover quickly from failures. Such characteristics and capabilities have made NAS devices popular in enterprise environments.

Unfortunately, storage scalability can become problematic when data needs outscale the capabilities of a single NAS device; providing redundancy across multiple separate NAS devices (as illustrated in FIG. 1A) can involve substantial configuration expertise. For instance, consider the scenario of responding to a drive failure. Typically, a redundant storage system attempts to restore lost bits and re-establish redundancy as quickly as possible. However, in some scenarios, depending on the application and load, the storage system may need to place higher priority on continuing to serve client requests with a specified level of performance, and hence may need to delay restoration efforts. Storage systems typically need to be architected very carefully based on expected client needs, application needs, and load characteristics.

FIG. 1A illustrates a set of clients (100-102) that are configured to access NAS devices (110-114). Note that management overhead typically increases in proportion with the

amount of storage available. For instance, as the number of supported applications and storage space increase, a storage system may need to grow to include a load balancer 120 between the clients (100-102) and the NAS devices (110-114), as illustrated in FIG. 1B. Load balancer 120 can explicitly partition applications and clients to a given NAS device, and then route requests accordingly. While initial NAS vendors primarily focused on speed and reliability, as storage needs have continued to grow NAS vendors have also begun to compete by including sophisticated system management solutions that facilitate adapting to different storage, performance, and failure scenarios.

FIG. 2 illustrates another network storage system that provides remote storage, but with a disk-level abstraction. In such an architecture, a computing device 200 manages metadata for a filesystem 202 locally, and then sends block-level read/write requests to a remote block storage device 204 via a storage area network (SAN) (e.g., by using the Internet Small Computer System Interface (iSCSI) or a Fibre Channel protocol). More specifically, block storage device 204 provides only a block storage interface, and is unaware of any filesystem structure associations (e.g., file names and/or structures) for the stored blocks. Such storage systems typically do not use transactional copy-on-write filesystems, and hence are not data-consistent.

Note that there is a significant distinction between filesystem-level operations and block-level (e.g., disk-level) operations. A filesystem typically serves as an intermediary between an operating system and one or more block-level devices. More specifically, a filesystem typically attempts to efficiently manage one or more block-level devices to provide more sophisticated storage services to an operating system. For instance, filesystems often manage disk blocks and metadata to provide structure (e.g., files and directories) and some notion of access rights and data consistency (e.g., via file lock operations) for an underlying block storage mechanism. Hence, filesystem-level operations provide a higher level of abstraction (e.g., a filename and an ordering associated with an underlying set of disk blocks) for the block storage mechanism.

Typically, a filesystem and an associated block storage device both operate in the context of the same computing device, and the block storage device has been specially initialized (e.g., formatted) to support the filesystem. Upon receiving a request for a filesystem operation (e.g., from an operating system and/or application), the filesystem determines and initiates a set of block-level operations needed to service the request. Hence, there is a notion of “filesystem-level information” (e.g., the level of information managed by the filesystem and received in a request for a file operation) and a separate notion of “block-level information” that is used when the filesystem operates upon the underlying block storage device. In the example of FIG. 2, the functionality of the filesystem and the underlying block storage are split across two different devices (computing device 200 and block storage device 204). As mentioned above, block storage device 204 provides only a block storage interface, and is unaware of any filesystem structure associations for the stored blocks. Block storage device 204 may store filesystem metadata on behalf of filesystem 202, but it is filesystem 202 that provides the higher level of abstraction to the operating system of computing device 200.

A number of “cloud-based storage” vendors (also sometimes referred to as “cloud storage providers”) attempt to simplify storage management by providing large-scale network storage solutions. Such vendors can leverage economies of scale to provide data centers with extensive data

storage capacity that can then be rented and accessed by clients, thereby allowing clients to offload storage management overhead and easily increase their data storage on an as-needed basis. However, cloud-based storage also includes another set of inherent risks and overheads. Storing data remotely (“in the cloud”) often increases access latency, and network failures and/or outages in cloud-based storage systems can prevent clients from accessing their data for substantial time intervals. Furthermore, multiple clients simultaneously accessing a shared data set in a cloud-based storage system may suffer from data consistency issues.

Consider a scenario where one remote client attempts to write a set of data to a cloud storage system, and a second remote client attempts to read the data that is being written. In some systems, a reader may not be able to see the existence of newly written file data until the entire write operation has completed (e.g., after the first remote client has closed the file). In other (non-data-consistent) arrangements, the reader may see and access the file, but because writes are stateless and potentially out-of-order (e.g., as in the Network File System (NFS) protocol), does not know which file sections have already been written, and hence may access a mix of valid data and garbage.

Embodiments of the present invention combine aspects of NAS capabilities and cloud-based storage capabilities to provide a high-capacity, high-reliability storage system that ensures that data can be accessed in a data-consistent manner. Providing Data Consistency in a Cloud Storage System

In some embodiments, a set of caching storage devices (referred to as “cloud controllers”) collectively cache, manage, and ensure data consistency for a set of data that is stored in a network storage system (e.g., a cloud-based storage system, which is also referred to as a cloud storage system). More specifically, one or more cloud controllers work together (e.g., as a federation) to manage a distributed filesystem with a global address space. Each cloud controller maintains (e.g., stores and updates) metadata that describes the file and directory layout of the distributed filesystem and the location of the data blocks in the cloud storage system. Each cloud controller can also cache a subset of the data that is stored in the cloud storage system. A cloud controller that writes (or modifies) data ensures that: (1) data changes are reflected in the cloud storage system; and (2) other cloud controllers in the system are informed of file and metadata changes.

Note that while the cloud storage system stores the data for the distributed filesystem, the cloud storage capabilities may be provided by an external vendor. An enterprise storing sensitive data in the distributed filesystem may not want this vendor to be able to access such data, and hence, the cloud storage system may be configured to store the distributed filesystem’s data in the form of encrypted storage volumes (referred to as “cloud files” or “drive files”). This configuration enhances data security, but also prevents the cloud storage system from actively assisting in ensuring data consistency and performing other operations that require knowledge of the data and data layout. More specifically, in some embodiments the cloud controllers fully manage the filesystem and manage data consistency, with the cloud storage system providing purely storage capabilities.

FIG. 3 illustrates an exemplary system in which a cloud controller 300 (e.g., a caching storage device) manages and accesses data stored in a cloud storage system 302. A request server 304 in cloud controller 300 may receive file requests from either local processes or via a network from a client 306. These requests are presented to a storage management system that includes a transactional filesystem 308 that manages a set of filesystem metadata 310 and a local storage system 312. In

FIG. 3, the filesystem structure defined by metadata 310 is illustrated as a tree of pointers that define one or more levels of directories and files residing in directories. Each file is described using a set of ordered metadata structures that indicate the set of disk blocks that contain the file’s data. A set of block records 314 in metadata 310 include pointer fields that indicate the location of the file data in a disk block 316 in local storage 312 (if the given block is currently being cached in the storage 312 of cloud controller 300), as well as the location of the file data in a cloud file 318. Note that disk blocks 316 and cloud files 318 may have substantially different sizes. For instance, cloud files might be much larger than disk blocks, and hence the data contained in a disk block 316 may occupy only a portion of a cloud file 320. Hence, one pointer field in block record 314 may consist of a block pointer (labeled “BLOCK PTR” in FIG. 3) that points to a specific disk block, while another field (labeled “CVA&OFFSET”) may include both a pointer to a cloud file (also referred to as a “cloud virtual address,” or CVA) and an offset into the cloud file.

Note that using a transactional filesystem in each cloud controller does involve some additional overhead. As described above, the transactional filesystem tracks modifications using delta encoding (instead of the more typical read/copy/modify operations used in many non-data-consistent filesystems). For instance, consider a 1 KB modification to an existing 3 KB file in a filesystem that supports 4 KB blocks. Using a traditional approach, the filesystem might read out the original 4 KB block, modify the block to reflect the updates, and then write the modified file back to the same block. In contrast, in a transactional filesystem, the original block is left unchanged, and the filesystem writes out the modifications and additional data to another empty 4 KB block. The metadata for the transactional filesystem is extended to support the notion of partial blocks and deltas (e.g., including one pointer that points to 3 KB of data in one block and another pointer that points to another block that contains 1 KB of additional data and a set of changes that should be applied to the initial 3 KB of data).

In some embodiments, using a transactional filesystem (e.g., transactional filesystem 308 in FIG. 3) in a cloud controller facilitates providing ongoing incremental snapshots of changes to a cloud storage system and other cloud controllers. More specifically, the transactional nature (e.g., the delta encoding of changes) can be extended to include a set of additional metadata structures that track recently changed data in the cloud controller. These additional metadata structures can then be used to quickly and efficiently construct compact snapshots that identify file metadata and file data that has changed due to recent write operations. Note that these snapshots do not involve copying a full set of metadata and/or every byte that was previously written for a file; instead, such snapshots compactly convey only the set of changes for the data set. Sending only a compact set of changes facilitates maintaining data consistency while minimizing the amount of data (and metadata) that needs to be transferred and processed. Sending frequent snapshots ensures that changes are quickly propagated to other cloud controllers and the cloud storage system.

In some embodiments, cloud controllers generate separate metadata snapshots and file data snapshots. Metadata is typically much smaller than file data, and is needed to access file data. Furthermore, each cloud controller is typically configured to maintain (and update) the full set of metadata, but only caches file data that is needed by local clients. Hence, uploading (or sending) a metadata snapshot separately means that the updated metadata will be more quickly available to other

peer cloud controllers. Each of these peer cloud controllers can then determine (e.g., based on client data usage and needs) whether to access the related file data associated with the updated metadata. Note that a cloud controller may still upload both metadata updates and file data updates to the cloud storage system, but may split them into different sets of cloud files (or both include the metadata with the file data as well as generate another separate, duplicative update that includes only metadata) so that other cloud controllers can access the two separately. In such an organization, a cloud controller might then send a message to other cloud controllers specifying the location of the stored metadata snapshot. Alternatively, cloud controllers may also be configured to send metadata snapshots directly to a set of peer cloud controllers.

Consider an example of a cloud controller receiving a request from a client to store a 10 GB file, in an environment where the network link between the cloud controller and a cloud storage system supports a transfer speed of 1 GB/minute and the cloud controller is configured to send a metadata snapshot every minute. Upon determining the scope of the file operation, the cloud controller can already allocate a set of corresponding disk blocks and cloud files, and generate a set of corresponding metadata that indicates the respective disk addresses and CVAs for the file's data blocks. The cloud controller then uploads the file data to the cloud storage system over a time interval (e.g., roughly ten minutes), and sends out metadata snapshots that indicate the existence and location of the data blocks. The cloud controller may convey a range of information about the data being uploaded to other cloud controllers depending on the level of transparency and availability desired for modified data. For instance, in some embodiments, the file remains accessible by clients via the originating cloud controller throughout the upload process. However, other cloud controllers that have received the corresponding metadata and seek to access modified data that has not yet been received by the cloud storage system may receive an indication that the data is not yet available, and that their access attempts should be re-tried at a later time (or after a specified time interval). Alternatively, in some instances, when a set of data has not yet been uploaded to the cloud storage system, a client (and/or cloud controller) that hence cannot yet access this data via the cloud storage system may be configured to gain access to the desired data by directly interacting with the cloud controller hosting the desired data. Such alternative access techniques may depend on the capabilities and topography of the network connecting the cloud controllers and cloud storage system.

In some embodiments, the originating cloud controller may propagate additional intermediate metadata that informs other cloud controllers as portions of the modified data become available in the cloud storage system. For instance, metadata snapshots may indicate files that are in the process of being uploaded, and include a field that indicates whether a given data block has been successfully stored in the cloud storage system. The cloud controller updates (and propagates) this metadata as it receives acknowledgments of receipt from the cloud storage system, thereby indicating that some of the data being uploaded is now already available in the cloud storage system. For example, immediately after first storing the 10 GB file locally, the cloud controller may have already reserved 10 GB of space in cloud files at a given set of CVA addresses (e.g., in the cloud storage system), but have not yet transferred any file data. A snapshot sent at this point includes metadata that indicates the existence of the file, but also indicates that none of the data is available in the cloud

storage system yet. After one minute, the cloud controller sends out another snapshot containing metadata that reflects the set of data that has already been transferred to (and been acknowledged as received by) the cloud storage system.

In some embodiments, each cloud controller maintains a set of structures that track snapshots and changes in metadata, and updates its local metadata to reflect updates from the rest of the distributed system. For instance, a cloud controller receiving the first snapshot from the above example may note the creation of a 10 GB file (as described in the above example), but then also determine that none of the associated data blocks is available yet. After receiving and processing the second snapshot, the receiving cloud controller determines the presence and location of the first GB of stored data that is now available. At this point, the receiving cloud controller may, if desired, use the received metadata to already download and access the available file data from the cloud storage system on behalf of a client. If, however, a client requests additional parts of the file that have not yet been stored in the cloud storage system (as indicated by the metadata in the most recent snapshot), the cloud controller can signal that the desired data is not yet available, and delay the access. More of the file data becomes available over time, as indicated by the subsequent snapshots.

Note that cloud controllers can use the detailed information received in snapshots to provide a range of data access and data consistency capabilities. More specifically, each cloud controller receives ongoing updates that identify valid data, and indicate how to find and access such data. If data is written to the cloud storage system out-of-order, this is reflected in the received snapshot(s), and the cloud controller (and/or a requesting client) can use such received snapshot information to determine how to proceed.

In some embodiments, cloud controllers may use stored snapshot data to provide access to different versions of a file. For instance, in the preceding example, a cloud controller may allow a client to already access the uploaded (and acknowledged) portions of a new file before the file has been completely uploaded to the cloud storage system. Similarly, the cloud controller may allow the client to access modified file data as it becomes available in the cloud storage system. Alternatively, in other scenarios, when an existing file is being modified, a cloud controller may be configured to present a previous version of the file to clients until the complete set of data for the modified version is available in the cloud storage system. In some embodiments, cloud controllers may maintain records of past snapshots to allow file accesses to be rolled back across multiple different versions, thereby allowing clients to view historical versions of files and/or the changes made to files over time.

In general, the disclosed techniques leverage transactional filesystem techniques and snapshots to ensure that only valid data can be accessed. While these techniques involve some additional complexity, they also provide an assurance of data consistency for a distributed filesystem that leverages cloud storage. The following sections describe additional aspects of storing and accessing data in the disclosed distributed filesystem.

Generating Snapshots and Cloud Files

A number of factors affect the performance of accessing data from a cloud storage system. In a typical computer data is stored locally on a disk, and a number of hardware and operating system mechanisms attempt to minimize the latency of reads and writes. For instance, processors and operating systems strive to load frequently used data into memory and multiple levels of hardware caches, thereby reducing the latency associated with reading data from disk.

11

Accessing data stored on a cloud storage system involves an additional set of latencies. For instance, in addition to normal disk latency, accessing a cloud storage system may involve additional latency due to network latency, network protocol handshaking, network transfer times, and delays associated with encryption or decryption. One of the challenges of a distributed filesystem is minimizing such latencies as much as possible.

One factor that can significantly affect the latency of data access in the described distributed filesystem is cloud file size. Overly small cloud files can result in higher network negotiation and transfer overhead. Conversely, overly large cloud files can result in large transfer delays; for instance, a cloud controller that needs only a small piece of data from a large cloud file that is serially encrypted may need to wait for the entire file to be downloaded and decrypted before it can access the desired data. Determining a reasonable cloud file size that maximizes throughput to and from the cloud storage system may depend on factors such as network link size and latency (e.g., transfer speeds), local filesystem block sizes (e.g., making the cloud file size a multiple of a local block size), and CVA pointer sizes or boundaries. Another trade-off involves determining whether to use fixed-sized or variable-sized cloud files. Variable-sized cloud files allow some level of customization to match network and application characteristics, but also involve additional complexity to manage the different sizes. Hence, in some embodiments the system reduces management overhead by using a single fixed cloud file size (e.g., 32 MB) throughout the cloud controllers and cloud storage system. Note, however, that the contents of each cloud file may vary based on the set of data currently being generated or modified. For instance, data blocks being stored for large files (e.g., larger than 32 MB in the case of 32 MB cloud files) may be split across two or more cloud files. Alternatively, if the current load involves storing data for multiple small files or making a large number of small file modifications, a corresponding cloud file may contain multiple user files and deltas. Note also that, in some embodiments, data and meta-data are always separated into different cloud files. In general, cloud controllers may use a range of techniques to stripe chunks of data across cloud files in an attempt to optimize subsequent accesses from such cloud files.

FIG. 4A illustrates the process of generating a cloud file for a snapshot in the context of the exemplary system of FIG. 3. As indicated by filesystem metadata 310, a file ("file X") includes a set of metadata 400 and a set of disk blocks (the highlighted blocks among disk blocks 316) that have been modified since a previous snapshot. During the snapshot process, cloud controller 300 freezes the pointers in the blocks, and determines the set of metadata and data that should be written out to cloud storage system 302. The modified data is then packaged into units that match the granularity of cloud files (e.g., into 32 MB segments), optionally encrypted, and then uploaded to cloud storage system 302.

Note that cloud files are also written to in an incremental, transactional fashion, to preserve data consistency. More specifically, new and modified file data is written to a separate cloud file, as in a transactional filesystem, to ensure that the consistency of previous file versions is preserved. Thus, an initial set of data for a given file is written to one cloud file, and later additions or modifications to the file detected by a subsequent snapshot are written to a new, different cloud file.

The filesystem metadata for each disk block includes information that specifically identifies the location and enables the lookup of the disk block in a cloud file. For instance, the metadata may include one or more of the following: a CVA

12

(cloud virtual address) that uniquely addresses the cloud file; the offset of the disk block in the cloud file; a physical and logical size for the disk block; the portions of the disk block that are valid; compression information; a checksum hash value or other checksum information; and information that indicates whether the disk block has already been successfully uploaded to the cloud storage system.

To ensure data consistency, cloud controllers need to ensure that each cloud controller assigns unique CVAs that create non-overlapping cloud files. More specifically, the cloud controllers need to collectively manage the global address space for the distributed filesystem. In some embodiments, each cloud controller is assigned a unique identifier, the collective set of cloud controllers are associated with a total amount of cloud storage space, and each cloud controller is pre-allocated a portion of the global address space. In such embodiments, a cloud controller can already allocate a cloud file in this pre-allocated address range at the time that it writes a new disk block, and store the CVA of the cloud file in the block's metadata. This organization ensures that there are no collisions in allocating cloud file addresses, and also ensures that even the first metadata snapshot for a new disk block already includes an accurate (future) location of the disk block in the cloud storage system. Note that the allocations of the global address space can be adjusted as needed over time if more data is created on a subset of the cloud controllers.

While most updates and file changes can propagate through the distributed filesystem via incremental snapshots, some data consistency issues can arise if multiple clients 30 accessing two different cloud controllers attempt to simultaneously write the same file. Downloading and importing snapshot data may involve some latency, and thus such conflicting operations may lead to race conditions and errors. Hence, in some embodiments, each file is associated with a cloud controller that "owns" (e.g., actively manages) the file. For instance, the cloud controller from which a file was first written may by default be registered (in the file block metadata) as the owner (e.g., the owning cloud controller) of the file. A cloud controller attempting to write a file owned by another cloud controller first contacts the owner with a request to lock the file. The owner can determine whether to grant or deny the lock request. In some embodiments, even if this request is granted, all write operations may be required to go through the cloud controller that owns the file (e.g., new data is written to the local filesystem of the owning cloud controller). Note that while every cloud controller actively manages a set of files, a given cloud controller may not need to continue to cache every disk block of files that it owns; once such blocks have been written to the cloud storage system, they may subsequently be cleared from the cloud controller to make space for other needed data. However, the metadata for all of the files in the distributed system is typically maintained in every cloud controller. In some embodiments, the system 45 may also include mechanisms for transferring ownership of files between cloud controllers (e.g., migrating file ownership to cloud controllers that are the primary modifiers of the file to reduce network latency).

Optimizing the Creation of Cloud Files

Note that a cloud controller may use a range of techniques to generate cloud files. For instance, one such technique may involve: (1) allocating one or more cloud-file-size memory buffers; (2) copying the file and metadata for the cloud file into a memory buffer; (3) encrypting the contents of the memory buffer; and (4) uploading the encrypted contents of the memory buffer to a cloud storage system as a cloud file. Note, however, that this technique involves allocating and

using additional memory buffers, and potentially performing a large number of data copy operations.

In some embodiments, a cloud controller generates an additional set of filesystem overlay metadata that allows existing file data and metadata to be virtually linked together into a cloud file view. For instance, the system can construct such overlay metadata when writing new blocks (and corresponding metadata). Alternatively, the cloud controller may instead generate such overlay metadata while traversing the filesystem to find changed data to include in the next incremental snapshot. Either way, unlike the above memory-buffer technique, this overlay metadata facilitates minimizing the use of additional resources by creating cloud files “in place” (e.g., without allocating additional memory buffers or additional copy operations); instead, a set of pointers point to the original blocks in the transactional filesystem that contain the modified data and metadata. Note that while such additional overlay metadata may involve some additional space and computational complexity, these additional needs are typically small compared to the space and copy overhead associated with other approaches.

When creating a snapshot, a cloud controller can access the overlay metadata to read, encrypt, and upload the cloud file to the cloud storage system. For instance, the overlay metadata may facilitate accessing the virtual cloud file via a special filesystem directory that presents a view of the disparate data blocks as a single, consolidated cloud file that can be read and transferred. In many scenarios the cloud controller primarily maintains overlay metadata for data that has not yet been written out to a cloud file; in some embodiments, once the data has been uploaded to the cloud storage system, the cloud controller clears the overlay metadata and begins generating new overlay metadata to track changes destined for a subsequent set of cloud files. In other embodiments, a cloud controller may maintain snapshot information and overlay cloud files locally for a longer time interval (e.g., until space constraints on the cloud controller prompt the removal of infrequently used data).

FIG. 4B illustrates a set of overlay metadata 410 and a virtual cloud file 412 in the exemplary system of FIG. 3. During operation, cloud controller 300 uses overlay metadata 410 to track the metadata 400 and data disk blocks (the highlighted blocks among disk blocks 316) that have been modified since a previous snapshot. During the snapshot process, cloud controller 300 reads and uploads the virtual cloud file 412 presented in the overlay metadata 410 into a cloud file in cloud storage system 302. Note that the transactional nature of cloud files can lead to substantial distribution of file data in a cloud storage system over time. For instance, files that are modified multiple times across multiple snapshots will be stored in different cloud files. Thus, a cloud controller that has flushed the file data (as described in more detail in the following sections) may need to download and access all of the relevant cloud files to reconstruct the file at a later time, which may involve considerable network bandwidth and time. Unfortunately, the initial cloud file for a given file is generated at the time that the first set of file data is written; at this time, little is known about the likely future access patterns for the file.

In some embodiments, a cloud controller attempts to optimize the placement of data into cloud files to reduce future access overhead. For instance, the cloud controller may strive to, when possible, store all blocks for a file in the same cloud file (e.g., assuming the size of the file and/or file modifications are smaller than the size of a cloud file). Toward this end, the cloud controller may place data into multiple cloud files in parallel, and avoid storing data for multiple files in the same

cloud file unless the complete set of data for some or all of the files will fit. A cloud controller may also perform additional file grouping based on user configuration and/or automatic analysis of file access trends. For example, users may be provided with a way to configure a policy that reflects anticipated file access patterns, groupings, and/or priorities (e.g., a user policy that indicates files with a certain extension are likely to be accessed together, and thus should be grouped together).

Note that some cloud files may be partially empty. For instance, a cloud controller that is generating a snapshot based on a time interval or a cloud controller that is placing data into multiple cloud files in parallel to optimize future read operations may not have enough data to fill a complete cloud file. In such scenarios, the cloud controller may simply write out the available data, and leave the wasted space to be reclaimed using a future reclamation operation. For example, in some embodiments a cloud controller may be configured to: (1) download file data spread across two or more cloud files; (2) reassemble the desired data into a new cloud file; (3) upload the new cloud file to the cloud storage system; and (4) distribute a metadata snapshot that updates the access information for the affected files. In some scenarios, such optimizations may be performed by the cloud controller that owns the files, and involve locking the files during the cloud file defragmentation process. In scenarios where locking files for a long interval is undesirable, such operations may involve: (1) maintaining the previous cloud files the final metadata snapshot has been generated; (2) determining that the file data has not changed during the download, reassembly, and upload operations; and (3) briefly locking the original cloud files and metadata sections until the system can confirm that the updated metadata has been successfully propagated to all of the cloud controllers in the system. If the file data being compacted has changed during the interval, the changes can be discarded and compaction can be re-tried at a later time.

Accessing Cloud Files and Managing Local Disk Layout

The previous sections disclose techniques for generating snapshots and uploading data to cloud files. Using such techniques, cloud controllers can treat the cloud storage system as an object store. Other cloud controllers receiving metadata updates can then access data from cloud files as needed. Furthermore, a cloud controller that has uploaded data can, if needed, flush data that has been uploaded from its local filesystem (e.g., “clear its cache”) to make space for other data that is more likely to be needed immediately. Note, however, that a cloud controller flushing data still keeps the accompanying metadata, so that the flushed data can be found and reloaded from the cloud storage system if needed again.

FIG. 4C illustrates a second cloud controller 420 that responds to a snapshot sent by the cloud controller 300 of FIGS. 3-4A. As described previously, cloud controller 300 generates a cloud file during a snapshot, and uploads the cloud file and the (incremental) metadata snapshot to cloud storage system 302 (as indicated by (1) in FIG. 4C). Upon receiving confirmation of the successful upload, cloud controller 300 then sends a notification to other peer cloud controllers (including cloud controller 420) that informs them of the availability of the new snapshot (as indicated by (2) in FIG. 4C). Cloud controller 420 then downloads the metadata snapshot from cloud storage system 302 (as indicated by (3) in FIG. 4C), and updates its local metadata accordingly (as indicated by (4) in FIG. 4D). After updating the metadata, cloud controller 420 can proceed to download any desired data from the corresponding cloud files. Note that in some

15

scenarios cloud controller 300 may also be configured to directly send the metadata snapshot as part of the notification (2) to cloud controller 420.

FIG. 4D illustrates the process of accessing data from a cloud file. At some point after receiving updated metadata from a snapshot (as described for FIG. 4C), cloud controller 420 receives a request from a client 421. The storage system on cloud controller 420 inspects its updated filesystem metadata 424, and determines that the request requires data that is not currently cached in local storage 426. The system then uses the lookup information in the block records of the metadata (e.g., the CVA and offset values) to determine the appropriate cloud file(s) to download. Cloud controller 420 then downloads (and decrypts, if necessary) the indicated cloud files, and uses the offset information in the metadata to unpack the desired contents of the downloaded cloud file(s).

In some embodiments, a cloud controller downloads a cloud file into a dedicated memory buffer, and operates directly upon this memory buffer to access the desired data. The cloud file format is easy to operate upon in memory, and the downloaded data can be accessed very quickly from memory. However, storing such blocks in memory also constrains memory use, and (depending on the application) client data demands may involve reloading and caching more data than can be stored in memory. Furthermore, operating upon downloaded cloud files purely in memory may require a cloud file to be re-downloaded if the cloud controller is power cycled. Hence, in alternative embodiments, the cloud file is unpacked and re-integrated into the local transactional filesystem of a downloading cloud controller (e.g., into the highlighted subset of disk blocks 428 in FIG. 4D). Integrating downloaded cloud file data into the local filesystem allows all filesystem accesses to operate in the same manner (e.g., as opposed to special memory-buffer accesses for downloaded cloud file data), and facilitates keeping the downloaded data persistent across power cycles.

Note that a cloud controller can choose the target disk blocks that will receive the downloaded data blocks; for instance, in the context of FIG. 4D, cloud controller 420 is not constrained to use the same set of disk blocks used by cloud controller 300 to store the same data. In some instances, this selection process may involve determining disk blocks containing data that has not been accessed recently (e.g., via a least-recently-used (LRU) policy), and flushing such data. Each cloud controller tracks the usage of its cached data blocks, and strives to cache data blocks that minimize that latency experienced by clients.

Some fragmentation issues can arise on cloud controllers over time, as cached data is flushed and new (and/or old) data is loaded from cloud files. Recall that cloud controllers typically maintain the full set of metadata, but flush actual data as needed. In general, as new data is created and written into files, target disk blocks are found to store the resulting data and metadata. To optimize reads and writes, a filesystem normally places metadata in close proximity to its referenced data. However, such organization can lead to fragmentation issues for a system in which data can be flushed (and reloaded), but metadata remains persistent. More specifically, if such data is stored interleaved and relatively continuously (e.g., to improve read access), the subsequent holes left by flushed data may be refilled with new metadata and data. Over time, these operations lead to substantial fragmentation, with small pieces of metadata strewn across the disk and a rarity of larger contiguous disk areas that can be used to store new file data. In some cases disk defragmentation techniques may be used to alleviate some of these issues, but defragmentation is

16

typically both time- and disk-intensive, and client data access performance may be degraded during the defragmentation process.

In some embodiments, the block allocation policy used in a cloud controller's transactional filesystem is altered to prioritize a selected set of disk sectors toward either data or metadata. More specifically, by dynamically weighting some disk blocks toward metadata, the filesystem can create dedicated, metadata areas on the disk that are distinct from their respective data blocks, and no longer interleaved on a per-file basis. While distinct, these metadata areas can still be allocated in close-enough proximity to the data blocks that they reference that both can be read without substantially degrading performance. When data is subsequently flushed, all of the disk blocks holding data are cleared, and new data and metadata can be written into the disk region; new metadata is written into the disk blocks weighted toward metadata, while the new data blocks can be stored into the nearby (flushed) disk blocks. Because metadata is typically much smaller than the actual file data (e.g., in many scenarios metadata is on the order of 0.1% of the size of the file data that it manages), this arrangement facilitates avoiding fragmentation across a large number of write/flush cycles.

Note that the amount of metadata in a distributed filesystem may, depending on the client and application load, grow significantly over time. Hence, cloud controllers may need to periodically adjust the number and weighting of disk blocks containing metadata, so that more blocks that store metadata are allocated in proximity to a set of blocks used to store file data. Note also that in some embodiments the filesystem may attempt to optimize read accesses by trying to store (where possible) disk blocks that have been loaded from cloud files in proximity to their metadata.

Pre-Fetching Cloud Files

As mentioned previously, cloud files are also written to in an incremental, transactional fashion. For instance, files that are written and/or modified across multiple snapshots may have data stored in different cloud files. Unfortunately, accessing data split across multiple cloud files can increase access latency substantially. Consider a client that sends a request to a cloud controller to access a data block in a file. The cloud controller inspects its current set of metadata, determines that the data block is not currently cached, downloads the corresponding cloud file containing the data block, and presents the data block to the client. The client may then request additional data blocks from the same file. In some scenarios, all of these blocks will have been stored in the same cloud file, and are now available in the cloud controller. If, however, some of these file blocks are in another cloud file, the client will have to endure another set of network, download, and processing latency as one or more additional cloud files are accessed. While an initial delay may be acceptable, recurring access delays may cause a noticeable delay and substantial user frustration.

In some embodiments, a cloud controller predictively prefetches additional cloud files in an attempt to reduce access latency. For instance, upon receiving a request to access a given data block for a file, a cloud controller may analyze the metadata for the file and then predictively pre-fetch other cloud files that contain other nearby data blocks (or even all other data blocks for the file, depending on the file size). Alternatively (and/or additionally), the cloud controller may also pre-fetch data for other associated files that are likely to be accessed in conjunction with the original file. In both situations, the cloud controller can traverse its stored set of

metadata to look up the physical locations (e.g., the CVAs and offsets) for cloud files that should be pre-fetched from the cloud storage system.

Note that the amount and range of pre-fetching performed may vary based on factors such as the available network bandwidth, cloud controller load, and types of accessing clients or applications. For instance, a basic optimization may involve pre-fetching data for other files in the same directory as the current file being accessed. More sophisticated pre-fetching techniques may involve receiving application (or user) feedback that specifies file associations and/or files that are likely to be needed soon. Cloud controllers may also be configured to track access patterns over time to determine files and file sections that have temporal locality. For example, consider a large (e.g., terabyte-sized) set of seismic data that is typically accessed sequentially. Cloud controller tracking may: (1) detect the sequential nature of the data; (2) make note of this behavior in the file's metadata; and then (3) perform selective pre-fetching upon successive accesses to ensure that each subsequent cloud file is pre-fetched, thereby providing latency-free file access for a client. Note that some of these techniques may also facilitate initially grouping commonly accessed files and/or data blocks into the same cloud file so that they can be accessed more efficiently at a later time.

FIG. 5 presents a flow chart that illustrates the process of pre-fetching data for a distributed filesystem. During operation, a cloud controller that maintains a set of metadata for the distributed filesystem receives a request to access a data block for a file (operation 500). The cloud controller traverses the metadata to identify a metadata entry that is associated with the data block (operation 510), and then uses this metadata entry to download a cloud file containing the data block from a cloud storage system (operation 520). While performing these operations, the cloud controller additionally determines that an additional cloud file in the cloud storage system includes data that is likely to be accessed in conjunction with the data block (operation 530), and proceeds to pre-fetch this additional cloud file from the cloud storage system (operation 540).

Receiving and Servicing Client Requests

Client systems typically use network protocols (such as the Network File System (NFS) and the Common Internet File System (CIFS) protocols) to access network-based storage systems. CIFS (also sometimes referred to as Server Message Block (SMB)) is a complex application-layer network protocol that includes many application-specific capabilities that blur the typical separation between filesystems and applications. When a user accesses a file on a client system using software (e.g., Microsoft Office) that supports CIFS functionality, the client system may send CIFS requests to the network storage system to inform that the file has been opened and to store specific data. The CIFS server receiving the request for the file operation recognizes the file type, and can provide application-specific support. For instance, the CIFS server may maintain a network connection, and after determining that only that client is accessing the file, instruct the client to continue to cache file data indefinitely. Later, upon detecting that another client is attempting to open the file, the CIFS server can instruct the first client to flush the cached file data, and provide collaboration capabilities to both clients. Such functionality can enhance the user experience, but can complicate the interface between clients and storage systems; crafting a high-performance implementation of the CIFS protocol can involve substantial effort and expense.

In addition to complexity, CIFS also suffers from a range of other drawbacks and/or limitations. CIFS does not allow

requests to be proxied; all requests need to be addressed directly to the hosting server, which locks the underlying data. CIFS is also a "chatty" protocol (e.g., CIFS generates substantial bi-directional status and update traffic). In a distributed environment, these characteristics can result in substantial load and latency issues.

Embodiments of the present invention combine cloud controllers with NAS capabilities and cloud-based storage to provide a high-capacity, high-reliability storage system that can be accessed from multiple front-ends via an application-layer network protocol (e.g., CIFS).

In some embodiments, a customized filesystem device driver in an operating system decouples filesystem functionality from an underlying block storage mechanism, thereby allowing filesystem-level information to be forwarded to another filesystem and/or data management mechanism. For instance, in some embodiments a customized filesystem device driver in an operating system may forward such filesystem-level request information to a range of network storage devices and/or distributed architectures that can provide enhanced data storage capabilities to client computing devices.

Operating systems often support several standard filesystems (e.g., the DOS filesystem, the New Technology File System (NTFS), and CDROM filesystems), but sometimes also include an open interface that facilitates accessing special devices and providing third-party interoperability (e.g., to support USB flash drives and filesystems associated with other operating systems, and to allow the development and support of future filesystems). In some embodiments, such interfaces can be used to create a filesystem device driver that emulates a local filesystem and storage device to the local operating system, but actually instead forwards filesystem-level request information to (and receives responses from) a non-block-level storage management system and/or filesystem.

In some embodiments, the disclosed forwarding techniques facilitate leveraging an existing implementation of a request server to reduce implementation overhead and complexity. For instance, as described above, implementing CIFS involves substantial effort and expense. Hence, some embodiments may provide CIFS capabilities by executing an operating system (e.g., Microsoft Windows Server 2008 Core, which includes embedded, high-performance CIFS server functionality with low memory usage) that can manage CIFS requests, but then use a customized filesystem device driver to extract and forward filesystem-level information that was received in CIFS requests. Note that this is distinct from techniques that direct file operation requests to a filesystem (such as NTFS) that then directly performs corresponding block-level operations upon a block storage device; forwarding filesystem-level information facilitates adding additional layers of capabilities and services prior to (or in place of) performing lower-level storage operations.

FIG. 6A illustrates a computing device 600 that receives and forwards requests for filesystem operations. Computing device 600 executes a request server 608 that receives requests for file operations from clients (610-612) in its computing environment 614. Request server 608 sends instructions to a filesystem device driver 616 to perform the requested file operations. However, instead of managing a disk drive and disk operations, filesystem device driver 616 can be configured to forward filesystem-level information associated with the request to a range of other devices and/or mechanisms. For instance, filesystem device driver 616 may be configured to forward filesystem-level request information to one or more of the following: a cloud storage system 302

that is outside local computing environment 614; a storage management system 632 on another computing device 630; and/or an NAS device 640. Note that NAS device 640 may comprise a range of capabilities and architectures. For instance, NAS device 640 may comprise a compute server that uses an NAS filesystem 642 (e.g., a transactional copy-on-write filesystem) and a range of local storage capacities 644 to handle network file requests.

In some embodiments, an NAS device serves as a cloud controller for the cloud storage system. In this role, the NAS device presents a standard request interface (e.g., CIFS) to clients, and uses local storage capabilities to cache the working data set that is being accessed in its local computing environment. The high-capacity cloud storage system stores the full data set for an enterprise, and serves as a backing store for the NAS device. This architecture combines the performance and interface of a local NAS device with the capacity of a cloud storage system, while reducing management complexity.

FIG. 6B illustrates a scenario in which storage management system 632, NAS filesystem 642, and storage 644 are co-located on an NAS device, cloud controller 601. For instance, filesystem device driver 616 may forward filesystem-level information from requests to storage management system 632, which can then use this information to determine whether file data should be stored (or accessed) in NAS filesystem 642 and storage 644 and/or cloud storage system 302. For instance, storage management system 632 may determine how to distribute and/or duplicate file information associated with the request between storage 644 and cloud storage system 302. The local working data set for an organization is usually relatively small (compared to the full enterprise data set), and hence can typically fit into a reasonably provisioned local storage 644 mechanism. From the client perspective, data access remains substantially similar to the simplest NAS device scenarios described above; computing device 600 serves as a single point of contact, no load balancer is needed to map applications of clients to specific NAS devices, and clients 610-612 are unaware of the interaction between storage management system 632 and cloud storage system 302. Note also that while request server 608 is not limited to receiving requests from local computing environment 614, request server 608 may also be configured to service requests for other clients outside of local computing environment 614. Similarly, in some scenarios one or more front-end computing devices 600 may be co-located with cloud storage system 302.

In some embodiments, filesystem-level information can be forwarded by executing an operating system with a desired server capability (e.g., Microsoft Windows Server 2008 Core) in a virtual machine. Note that, as above, this “guest” operating system does not execute in a stand-alone configuration (e.g., with an underlying NTFS filesystem and disk drive), but instead can be configured to forward filesystem-level information (e.g., CIFS requests) to (and receive responses from) an underlying storage management system that provides data-consistent capabilities. For instance, a customized filesystem device driver in the guest operating system can forward request information to (and receive responses from) a storage management system in the host operating system. Note that such forwarding behavior in the guest operating system is distinct from typical storage operations for a guest operating system. Typically, upon receiving a request for a file operation, a guest operating system accesses an associated (virtual) filesystem and outputs a block-level storage request that is received and handled by the host operating system; hence, no filesystem-level information is sent to the host

operating system. In contrast, in the described embodiments, the guest operating system forwards filesystem-level information, not block-level information, to the host operating system.

FIG. 6C illustrates a cloud controller 602 that contains all of the above-described capabilities. More specifically, cloud controller 602 receives requests directly from clients, and also serves as a front-end to remote cloud storage system 302. Cloud controller 602 includes a host operating system 604 that executes a guest operating system 606 in a virtual machine. Guest operating system 606 includes a filesystem device driver 616 that forwards requests for filesystem operations. A request server 608 (e.g., a CIFS server) in guest operating system 606 receives requests for file operations from clients (610-612) in its local computing environment 614. Request server 608 sends instructions to filesystem device driver 616 to perform the requested file operations. However, instead of managing a disk drive and disk operations (and/or using a virtual filesystem to send block-level requests to host operating system 604), filesystem device driver 616 is configured to forward filesystem-level information associated with the request to a storage management system 618 in host operating system 604 (e.g., via a virtual device interface in host operating system 604). As in FIG. 6B, storage management system 618 then determines how to distribute and/or duplicate file information associated with the request between local storage 620 and cloud storage system 302.

FIG. 7A presents a flow chart that illustrates the process of 30 forwarding filesystem-level information. During operation, a computing device receives a request for a file operation from a client computing device (operation 700). In response, the operating system of the computing device issues one or more filesystem operations for the request (operation 710). A filesystem device driver receives and extracts filesystem-level information from these filesystem operations (operation 720), and then forwards the extracted filesystem-level information to a storage management system (operation 730).

FIG. 7B presents a flow chart that illustrates the process of 40 using a guest operating system to forward filesystem-level information. During operation, a cloud controller that serves as a front-end for a cloud storage system executes a guest operating system on a virtual machine (operation 740). When this guest operating system receives a request for a file operation from a client computing device (operation 750), a filesystem device driver for the guest operating system forwards filesystem-level information associated with the request to the host operating system of the cloud controller (operation 760). The host operating system then uses this forwarded information to manage a file associated with the request in the cloud controller and in a cloud storage system (operation 770).

In some embodiments, the customized filesystem device driver extracts, tracks, and forwards client file interactions on 55 a per-file and a per-directory basis. More specifically, semantic filesystem-level information included in the application-layer network protocol (e.g., CIFS) is forwarded by the filesystem device driver to a storage management system. This semantic information can include, but is not limited to: a file name; a file type; a requested file operation (e.g., a read, write, or update operation); a set of application information associated with the file; one or more users accessing the file; and security information for the file. Cloud controllers can use this information to determine whether a file and its associated 60 information should be cached locally and/or forwarded to the cloud storage system (or other devices accessing the cloud storage system, as described below). For instance, the storage

management system may know that certain files will be duplicated and/or shared shortly after being modified, and hence may ensure that such files are both cached locally and forwarded to the cloud storage system to facilitate the expected duplication operation.

In some embodiments, decoupling a filesystem from underlying block storage devices facilitates transparently changing (e.g., either increasing or decreasing) the amount of storage space accessible by clients. Operating systems typically assume that filesystem device drivers always manage fixed-size volumes; storage devices normally have a fixed size, so this usually is not an issue. However, one of the benefits of using cloud-based storage is the ability to easily increase data capacity on demand. For instance, for the above-described scenario where a cloud controller caches data for a cloud storage system, the amount of space available to clients can be increased by leasing additional space in the cloud (network) storage system and communicating the change to clients as needed (e.g., upon request) through the filesystem device driver. Hence, in such embodiments the customized filesystem device driver and the disclosed caching architecture substantially simplify adjusting data storage capabilities. In contrast, expanding traditional storage systems typically may involve shutting down a storage device, physically adding additional storage devices, and then reconfiguring the entire storage system to accommodate the added storage space.

In some embodiments, volume size can be changed transparently regardless of any pending client operations. In alternative embodiments, the presence of some active filesystem connections and/or operations may require some or all connected clients to be disconnected during volume size changes (e.g., preventing some clients from actively accessing files), which may be infeasible or inconvenient. Hence, in some embodiments, the filesystem device driver may be configured to claim an initial fixed size that substantially overstates the expected amount of storage, to prevent future resizing logistics. The allocated portion of the cloud storage system may initially provide only a small subset of this claimed storage size, but then subsequently be dynamically expanded as needed. In some embodiments, such size configurations may be accompanied by user quotas to prevent storage space wastage.

Note that a filesystem device driver provides a higher level of abstraction than techniques that attempt to analyze disk-block-level traffic (e.g., in the disk-level remote storage system illustrated in FIG. 2). More specifically, such techniques attempt to recreate filesystem information by reverse-engineering block-level storage transactions. However, because these storage transactions typically do not include higher-level (e.g., file- and directory-level) information, such attempts typically involve substantial effort, and do not provide the same level of tracking capabilities. In contrast, a customized filesystem-level device driver facilitates forwarding reliable high-level semantic information without additional processing overhead and without modifying the guest operating system.

In some embodiments, the described techniques provide filesystem-level proxy functionality. In many situations, proxies are implemented at the abstraction level of networking protocols, but this becomes more difficult as network protocols become more complicated (e.g., by adding application-specific information, as in CIFS). Hence, instead of re-implementing a complicated network protocol, some of the disclosed embodiments create a proxy at another, simpler layer by using a customized filesystem device driver that extracts and “tunnels” (e.g., forwards) filesystem-level infor-

mation to another storage management system. Note that a filesystem-level proxy can also provide additional benefits. For instance, by emulating a local filesystem and storage device, the disclosed techniques can also overcome restrictions imposed by certain resource-intensive applications (e.g., certain databases, email server products, and/or data protection managers) to only use local storage devices. Using Multiple Cloud Controllers to Access Cloud-Based Storage Via CIFS

The previous section described some possible architectures for cloud controllers. As described previously, two or more cloud controllers may work together to collectively manage and access a shared set of files that are stored in a cloud storage system.

FIG. 8 illustrates multiple cloud controllers 800-802 (as individually illustrated in FIGS. 3-6C and described above) that collectively manage data in cloud storage system 302. Both cloud controllers 800-802 support application-layer network requests (e.g., CIFS requests) from their respective clients, and then collectively ensure data coherency and access performance for the shared data. As described previously, storage management systems in cloud controllers 800-802 incorporate aspects of a transactional copy-on-write filesystem, thereby ensuring that file operations are data-consistent and that the system can quickly recover from crashes.

Note that the described architecture can overcome limitations in an application-layer network protocol. As described above, CIFS does not allow requests to be proxied, and requires that a single hosting server manage the underlying filesystem. However, the disclosed filesystem forwarding mechanisms provide a level of abstraction where each guest operating system assumes it is the sole manager of the shared data. The underlying storage management systems in the host operating systems can pass application-level request information to one another to ensure that consistency is maintained. Thus, the disclosed techniques leverage existing CIFS implementations (via the guest operating system), but also extend CIFS capabilities to allow multiple CIFS servers to share access to underlying data. Note that the storage management systems may not need to understand all of the application-level information being received via CIFS requests; in some instances, the cloud controllers may simply forward such information to each other. However, for some data access scenarios (e.g., multiple clients accessing a shared file via different gateways), the storage management systems in each cloud controller may need to track and actively manage file operations to ensure that data consistency is maintained for the underlying data. Note also that receiving and filtering CIFS requests at the cloud controllers can reduce the amount of protocol chatter that travels over wide-area network links. For instance, a storage management system in a local cloud controller may be able to resolve some subset of CIFS requests, thereby eliminating the need to forward associated request information to the other components of the distributed storage system and reducing request latency.

In some embodiments, an additional network storage system may be used to provide another level of redundancy (and ensure high availability). For instance, in FIG. 8, either the storage management systems in cloud controllers 800-802 and/or cloud storage system 302 may be configured to mirror updates (e.g., also send metadata and data snapshots) to a mirror storage system 804 which is primarily used in failure situations. For instance, if cloud storage system 302 were to crash or become unavailable due to a network partition, cloud controllers 800-802 could be configured to temporarily use mirror storage system 804 as their backing store. As an emer-

gency backup, mirror storage system 804 may include fewer resources (e.g., a smaller network link and/or less storage capacity) than cloud storage system 302, but still ensure that availability and performance guarantees can be met. Note also that cloud controllers may be deployed as mirrored pairs, to ensure high availability across cloud controller failures. The following sections discuss redundancy and recovery options in more detail.

Disaster Recovery and Business Continuity

Some embodiments of the disclosed system architecture support the rapid recovery and/or replacement of cloud controllers. For instance, a new (or recovering) cloud controller immediately begins reading and processing the available metadata snapshots (e.g., in the case of a recovering cloud controller, the set of incremental metadata snapshots written to the cloud storage system since the cloud controller went offline). Because metadata is relatively small in comparison to the actual file data, this process can be performed more quickly than transferring a full data set. A cloud controller with relatively up-to-date metadata can immediately begin processing client requests; while the cloud controller may not yet have the requested data (or any data at all) cached, once it has retrieved a sufficient set of metadata it can already identify and access the cloud files containing requested file data.

In some embodiments, one or more additional cloud controllers serve as backups for an existing set of cloud controllers. These additional backup cloud controllers can take over the load of damaged and/or failing cloud controllers to provide disaster recovery and business continuity.

FIG. 11A illustrates a set of cloud controllers 1100-1112 that manage and access data stored in a cloud storage system 302. Backup cloud controller 1100 serves as a “hot backup” for cloud controllers 1102-1112. During operation, cloud controllers 1102-1112 write incremental snapshots containing new metadata and data to cloud storage system 302. Cloud controllers 1102-1112 continuously receive incremental metadata snapshot updates (e.g., either from cloud storage system 302, as shown, or directly from the other cloud controllers), and update their local metadata with these updates to maintain a current view of the data stored in the distributed filesystem. Cloud controllers 1102-1112 also download data (e.g., stored in cloud files) from cloud storage system 302 in response to data requests from a set of clients 1114-1130.

Note that in FIG. 11A backup cloud controller 1100 is not associated with any clients, but is instead configured to take over the load of another failing cloud controller. In some embodiments, backup cloud controller 1100 does not (in its backup role) serve data to any clients; instead, backup cloud controller 1100 primarily receives and synchronizes the metadata updates sent by cloud controllers 1102-1112 to maintain a current view of the data stored in the distributed filesystem. In some alternative embodiments, backup cloud controller 1100 may also be configured to assist with serving client data requests during times of exceptional system load.

In some embodiments, a backup cloud controller may also be configured to predictively download some of the data created and/or cached by the other cloud controllers of the system, in an attempt to download data that is in the working set of other cloud controllers. While a backup cloud controller that is provisioned similarly to the other cloud controllers can only store a partial subset of the full caches of the combined set of other cloud controllers, caching the full set of metadata and at least some of the current data actively being operated on in the system can be of benefit if the backup cloud controller needs to take over the role of a failed cloud controller.

For instance, a backup cloud controller that is configured to serve as a “hot backup” may continuously load the (most-

recently-created) data that was most recently uploaded to the cloud storage system. Due to temporal locality, recently created data is very likely to be accessed again, so in some scenarios caching such data in a backup cloud controller may be an optimal caching choice. For example, consider a scenario in which a set of globally distributed cloud controllers collectively manage data stored in a cloud storage system. Due to geographic constraints, only a subset of the cloud controllers may be active (e.g., actively servicing client requests) at a time, thereby allowing a backup cloud controller using the above-described technique to cache a larger portion of each active cloud controller’s active data set. In the most extreme scenario, if only one cloud controller is active, the backup cloud controller may mirror a substantial percentage of the working set of that one active cloud controller. If an active cloud controller fails, the backup cloud controller’s cache is already loaded with much of the recent data. Alternatively, if an inactive cloud controller were to fail, the backup cloud controller has more time and leeway (due to the failing cloud controller’s current inactivity) to transparently load a data set that more closely matches the working set of the failed cloud controller.

Alternatively, if constantly downloading the most-recently-uploaded data to the backup cloud controller incurs too high a bandwidth cost, the backup cloud controller may instead be configured to periodically update the local cache (e.g., hourly). Other options include tracking the most-commonly-accessed set of files in each cloud controller (and sharing this tracking information with the backup cloud controller, which then caches that specific data) and/or allowing system users and/or administrators to select the set of data that is cached on the backup cloud controller (e.g., using locality policies, as discussed in a following section).

FIG. 11B illustrates the system of FIG. 11A after the failure of a cloud controller, 1102. Backup cloud controller 1100, which was previously primarily synchronizing metadata (and perhaps predictively downloading some of the data cached by cloud controllers 1102-1112, as described above), takes over the load of failed cloud controller 1102, and begins servicing requests from clients 1114-1116. More specifically, because of its ongoing metadata synchronization efforts, backup cloud controller 1100 can already provide clients 1114-1116 instant access to any needed metadata, and can either serve locally cached data or access the cloud storage system as needed in response to client data requests. Note that a backup cloud controller will generally be configured to stop performing general caching efforts after taking over the role of a failed cloud controller; at this point, the backup cloud controller typically manages its cache in a manner that optimizes the performance of the data requests received from its clients.

In some embodiments, a backup cloud controller may be located at a location that is distinct from any other cloud controller, to ensure that a local disaster or issue does not also affect the backup cloud controller. However, in alternative embodiments, and in embodiments that include multiple backup cloud controllers (which might also serve in a load-balancing capacity when needed), some or all of the one or more backup cloud controllers may be co-located with active (non-backup) cloud controllers. Note that in scenarios that include multiple backup cloud controllers, individual backup cloud controllers may be associated with (and cache data for) a distinct subset of the active (non-backup) cloud controllers.

In some embodiments, backup cloud controllers (and/or other cloud controllers or other system entities) may be configured to automatically detect the failure of a cloud controller and perform a set of network and/or client reconfiguration options that automatically reroute requests from orphaned

25

clients to a backup cloud controller. In scenarios where multiple backup cloud controllers are available, backup cloud controllers may also be configured to periodically track the network latency and bandwidth between each backup cloud controller and non-backup cloud controller; such information can assist in determining which backup cloud controller can provide better service to the orphaned clients. In alternative embodiments, rerouting requests from orphaned clients to a backup cloud controller may involve manual intervention by a system administrator (e.g., explicitly changing a distributed filesystem mapping in a local network to route client requests to the backup cloud controller).

Note that transitioning from a failed cloud controller to a backup cloud controller may not be completely transparent from the perspective of orphaned clients of the failed cloud controller. For example, a client may have been transmitting data to be written to the failing cloud controllers, or the failing cloud controller may have been preparing (or even already been transmitting) data and/or metadata updates to the cloud storage system. Data in transit to or from a failed cloud controller should typically be considered lost; a primary consideration for the distributed storage system is that no data corruption occurs at the remaining cloud controllers or the cloud storage system.

In some embodiments, depending on the desired failure characteristics, the distributed storage system may have different modes of operation. For instance, in some modes of operation cloud controllers may perform strict ordering of updates; e.g., always upload an incremental data snapshot before its associated incremental metadata snapshot, to ensure that the global system metadata isn't updated until the uploading cloud controller has confirmed that both the data and metadata have been successfully received by the cloud storage system. For example, the cloud controller may calculate checksums (e.g., MD5 checksums) prior to uploading each snapshot, and confirm that these checksums match checksums generated by the cloud storage system upon receiving and successfully storing the snapshots. In the worst case, such strict ordering may result in the loss of some storage space in the cloud storage system (e.g., if the cloud controller fails after uploading snapshots but prior to sending notification of the metadata and/or data updates to other cloud controllers); this storage space can be recovered during cloud-controller recovery or via administrator actions.

In alternative embodiments, incremental metadata snapshots are uploaded first (to propagate the awareness of new data throughout the system as quickly as possible), with data snapshots following. In such embodiments, a cloud-controller failure that occurs between the upload of the metadata and data snapshots may result in metadata references to unavailable cloud files. In such situations, data corruption issues may need to be resolved by administrator actions or by inspecting log data to revert the metadata uploaded by the failing cloud controller.

FIG. 12 presents a flow chart that illustrates the process of providing disaster recovery for a distributed filesystem. During operation, two or more cloud controllers collectively manage the data of a distributed filesystem (operation 1200). More specifically, whenever each cloud controller receives new data from a client, it sends incremental metadata snapshots for the new data to the other cloud controllers, and sends an incremental data snapshot containing the new data to a cloud storage system. A backup cloud controller associated with the distributed filesystem is also configured to receive each (incremental) metadata snapshot (operation 1210), such that, upon determining the failure of a cloud controller, the

26

backup cloud controller can immediately begin receiving data requests from clients associated with the failed cloud controller (operation 1220).

Multiple Cloud Storage Providers

- 5 Cloud storage providers may be differentiated (and selected) based on a range of characteristics including, but not limited to:
 - capacity;
 - cost per unit of storage, network transfer, access, and/or processing activity;
 - request response/performance (e.g., internal latency when servicing a request);
 - frequency and/or severity of outages (with less reliable cloud storage providers typically offering cheaper pricing to make up for the potential unreliability);
 - location (which may involve political concerns) and network latency; and
 - level of replication/redundancy and number of points of presence (POPs) (e.g., some higher-cost cloud storage providers may immediately internally replicate newly stored data to multiple POPs at different geographic locations, thereby ensuring very high availability and low-latency access from anywhere in the world).

Choosing an appropriate cloud storage provider for a distributed filesystem may also depend on determining an anticipated data set and access patterns.

In some embodiments, multiple cloud storage providers may also be used to provide high availability and disaster recovery. For example, data may be mirrored across multiple cloud storage providers to ensure that data is still available to clients even if there is an outage in one of the cloud storage providers. Alternatively, multiple different cloud storage providers that provide different tiers of performance (and have different cost structures) may also be combined to balance performance and overall system cost.

FIG. 13A illustrates a set of cloud controllers 1300-1302 that manage and access data mirrored across multiple cloud storage systems (e.g., multiple cloud storage systems hosted by multiple distinct cloud storage providers). During operation, cloud controllers 1300-1302 write incremental data and metadata snapshots to both cloud storage systems 1304 and 1306. Load (e.g., cloud controller read requests) may be balanced between the two cloud storage systems depending on a number of desired criteria. For instance, if cloud storage systems 1304 and 1306 typically provide comparable performance, but cloud storage system 1304 has a lower per-access cost because it has occasional outages, the cloud controllers may be biased to primarily read from cloud storage system 1304, and only access cloud storage system 1306 when 1304 is suffering from an outage. In another example, cloud storage system 1304 may be more reliable and cheaper than 1306, but have worse performance due to fewer POPs (and, as a result, higher network latencies); in such a scenario, the cloud controllers might be biased to optimize performance by primarily reading data from cloud storage system 1306. Note that such choices are mostly transparent to the clients of cloud controllers 1300-1302; clients continue to access cloud controllers as previously, but may notice different levels of file access performance depending on the underlying cloud storage system choices (e.g., the cloud controllers' caches may hide some, but not always all, of the performance and latency aspects of an underlying cloud storage system).

In some embodiments of a distributed filesystem with data mirrored across multiple cloud storage systems, a cloud controller may be configured to immediately write a cloud file to a first cloud storage provider (thereby allowing the data to be propagated to other cloud controllers), but then delay the

transfer of the cloud file to the mirror to a time when network bandwidth is cheaper. In such embodiments, the cloud controller may be specially configured to ensure that the cached local copy of the data in the cloud file is not flushed until after it has been mirrored to the second cloud storage provider.

FIG. 13B illustrates an exemplary CVA format **1310** that facilitates identifying and locating cloud files across different cloud storage configurations. As described previously, accessing a block from a cloud file involves using a CVA to download the needed cloud file and then using the offset into the cloud file to access a specific target block. Fields in exemplary CVA format **1310** include, but are not limited to: a cloud controller ID (CCID) that uniquely identifies the originating cloud controller for the respective cloud file (e.g., the cloud controller that uploaded the cloud file); a filesystem ID (FSID) that identifies one or more cloud storage providers and/or other storage options; a snapshot ID (SSID) that uniquely identifies the updates of each cloud controller (e.g., each CCID+SSID combination in the system is unique); and a file ID (FILEID) that identifies the cloud files within each snapshot (e.g., each FILEID for a given SSID is unique). Note that ensuring the uniqueness of CVAs can be quite important for avoiding data corruption, because cloud storage providers typically serve as “black boxes” to the cloud controllers. For instance, a cloud controller may not know anything about the organization or layout of data in a cloud storage system, but may simply use a CVA to generate a unique file name and/or identifier for a cloud file to be stored to the cloud storage system. For example, in the context of the exemplary CVA format **1310** of FIG. 13B, a CVA may be converted to an object (file) name such as:

“data-<CCID>-<FSID>-<SSID>-<FILEID>.snp”

By using such naming conventions, a CVA can easily be used to determine the identifier for the cloud file; during a subsequent access, the same (or another) cloud controller can use the same CVA to generate and present the same file name to the cloud storage system and receive in return the cloud file that was stored in association with that CVA. Hence, the fields of CVA format **1310** need to be sufficiently provisioned to ensure that two cloud files are never written to the same cloud storage system for the same identifier; for example, CVA format **1310** may include a 64-bit SSID field to ensure that there will never be duplicate snapshot numbers for a cloud controller.

In some embodiments, an FSID field can be used to support multiple cloud storage providers. For instance, part of the FSID field may form a bit vector that indicates one or more cloud storage providers that are currently storing an associated cloud file. In another example, a cloud controller may use a value stored in a CVA’s FSID field to perform a lookup in a table of cloud service provider credentials. This table may include a list of cloud storage providers that are currently storing the cloud file, as well as “cloud account” information (e.g., information identifying a specific user account at a cloud storage provider and credentials that are needed to access that user account). Note that in addition to accessing different cloud storage providers, a cloud controller may also be configured to access different cloud accounts at the same cloud storage provider (e.g., different user accounts with different configurations and/or levels of service at the same cloud storage provider). FSIDs may also be used to support other filesystem-level features, such as storage quotas.

In some embodiments, data in the distributed filesystem may be split across multiple different cloud storage providers based on factors such as access frequency, age, and cost. For instance, new data may initially be written to a higher-cost

cloud storage provider that instantly replicates the stored data across multiple POPs; this wide initial distribution allows other cloud controllers requesting the new data (and metadata) to download it quickly. At some subsequent time, data that is no longer frequently accessed may be migrated into a cheaper lower-tier cloud storage provider (e.g., a cloud storage provider with higher latency and lower cost) and deleted from the first cloud storage provider. Such moves may be performed asynchronously and as background operations to ensure that users accessing cloud controllers and data are not adversely affected. For example, data may be migrated to the lower-tier cloud storage provider at a time of day when the load of the distributed filesystem and the cost of network bandwidth are both lower.

FIG. 14 illustrates the process of migrating a cloud file **1400** from a first cloud storage system **1402** to a second cloud storage system **1404**. Upon determining that all of the contents of cloud file **1400** are no longer being frequently accessed, cloud controller **1406** downloads cloud file **1400** from cloud storage system **1402** and then writes cloud file **1400** to cloud storage system **1404** (operation 1 in FIG. 14). Alternatively, if the cloud storage providers support direct transfers, cloud controller **1406** may instead be able to instruct cloud storage system **1402** to copy cloud file **1400** directly to cloud storage system **1404** (operation 2 in FIG. 14), thereby saving the overhead (and additional network bandwidth) involved in having cloud controller **1406** be an intermediary for the transfer.

In some embodiments, multiple factors are considered prior to migrating data between cloud storage providers. For instance, in some scenarios deciding whether to migrate a given cloud file may involve considering: the cost of storage at both the source and target cloud storage providers; a variable network bandwidth cost and/or the network bandwidth cost for the transfer; the access frequency and/or history for the contents of the cloud file; the potential performance impact of moving the cloud file to a lower tier; and the load of one or more cloud controllers. In some scenarios, cloud controllers actively monitor the cloud files and/or data files that they “own” (e.g., created) to determine how frequently they are accessed, and then use this information to determine candidates for migration. For example, a cloud controller may track the most recent access (e.g., the last read time) for individual blocks in its local persistent read cache (and/or in a persistent read cache that is distributed across multiple cloud controllers). After the last block for a cloud file is evicted from the read cache (e.g., due to not being used recently), the cloud controller may initiate a counter; if no blocks from the cloud file are used before the counter reaches zero, the cloud file becomes a candidate to be moved to a lower tier. Alternatively, the cloud storage system may be configured to track how often each given cloud file is accessed; the cloud controller that created a drive file may also check this access log to determine data that is no longer frequently used. Note that the above scenarios keep a cloud file in the higher tier cloud storage system if any of its blocks are still being actively used. In other scenarios, such decisions may be more complex (e.g., migration choices may also be affected by user-defined locality policies and/or cost-performance trade-offs).

In some embodiments, migrating a cloud file to a different cloud storage provider and deleting the copy from the previous cloud storage provider involves some additional logistical operations and/or policies to ensure that cloud controllers can still access the cloud file as needed. For instance, in one system metadata may not be updated to reflect migrations, and cloud controllers are configured to incrementally check

cloud storage providers in priority order (using the same CVA as an identifier) until the target cloud file is found. In other scenarios, the metadata for files may be updated to indicate that the cloud file has moved. Such indications may range from changing a bit field that is associated with a CVA (but not included in the actual CVA itself) to actually creating a new CVA that correctly references the cloud file on the new cloud storage provider. In the latter scenario, updating the metadata may be a multi-step operation. For example, a cloud controller may: (1) determine that a given cloud file should be migrated; (2) determine a new CVA for the cloud file at the new cloud storage provider; (3) upload the cloud file to the new cloud storage provider using the new CVA as the identifier; (4) upon receiving confirmation of receipt from the new cloud storage provider, update the metadata for all of the file blocks in the migrated cloud file to point to the new CVA (ideally without otherwise modifying the access history or other characteristics for the file blocks); (5) distribute an incremental metadata snapshot to the other cloud controllers; and (6) after enough time has passed to ensure that the other cloud controllers have updated their metadata, delete the cloud file from the old cloud storage provider.

FIG. 15 presents a flow chart that illustrates the process of distributing data for a distributed filesystem across multiple cloud storage systems. During operation, two or more cloud controllers collectively manage the distributed filesystem data that is stored in one or more cloud storage systems (operation 1500); the cloud controllers cache and ensure data consistency for the stored data. Whenever each cloud controller receives new data from a client, it sends an incremental metadata snapshot for the new data to the other cloud controllers and an incremental data snapshot containing the new data to a cloud storage system. Data stored in the distributed filesystem can be distributed across two or more cloud storage systems to optimize performance and/or cost for the distributed filesystem (operation 1510).

Archival Cloud Storage Providers

In some embodiments, a distributed filesystem can also leverage an “archival” cloud storage system. Archival cloud storage systems typically provide storage for a very low cost, but are equipped with a limited set of storage capabilities that are geared toward storing data that is infrequently accessed. For instance, archival cloud storage systems may be architected such that new data can be received and written immediately, but read requests for stored data sometimes involve potentially substantial delays (e.g., sometimes on the order of multiple hours, or longer). Such relaxed service constraints allow the cloud storage provider to optimize its data center to substantially reduce costs (e.g., by receiving data in a rotating set of active servers and powering down all other servers and storage devices completely, thereby dramatically reducing power and cooling costs); storage costs for archival cloud storage systems may be one or more orders of magnitude cheaper than non-archival storage systems. While such access constraints may not be appropriate for data that is being actively accessed by users, such storage capabilities can provide low-cost storage options for certain sets of archival data in a distributed filesystem.

In some embodiments, data writes can be mirrored to an archival storage system for disaster recovery. In such embodiments, writes can be mirrored as described previously (for multiple tiers), but the archival storage system would typically only be read if a primary (non-archival) cloud storage system were to be irretrievably lost (e.g., if the primary cloud storage provider were to go out of business or suffer from a substantial national disaster). In such arrangements, data might be sequentially read out of the archival storage system

(e.g., in order of priority) and then immediately written into another non-archival cloud storage system.

In some embodiments, the distributed filesystem collects and writes a set of archival data that is being retired from active use to an archival cloud storage system. This archived data will typically no longer be directly accessible by cloud controllers, but instead would need to be recovered by an administrator of the distributed filesystem.

As described earlier, cloud controllers notify each other whenever new data is created in the distributed filesystem. More specifically, cloud controllers periodically generate snapshots that reference every block, including both data and metadata, that is valid and stored in the cloud storage system at the time that the snapshot is taken. These snapshots are stored using data structures that allow each cloud controller to determine the set of new data and metadata that have been created in a recent time interval. Each cloud controller compares its two most recent snapshots (e.g., snapshot A and a later snapshot B) to determine the incremental difference between the two snapshots. For instance, data blocks that were created after snapshot A but before snapshot B (and are still referenced by snapshot B) are considered part of this incremental difference. Furthermore, any data blocks that were freed (e.g., deleted) between the snapshots are also considered part of this incremental difference. This incremental difference can be logically separated into incremental differences in metadata (e.g., new metadata created to reference newly created file data blocks) and incremental differences in data (e.g., the actual newly created file data blocks). In this disclosure, incremental differences in metadata are referred to as incremental metadata snapshots, and incremental differences in data are referred to as incremental data snapshots.

Each cloud controller stores both sets of incremental differences (e.g., the incremental metadata snapshot and the incremental data snapshot) in a cloud storage system. After uploading an incremental metadata snapshot to the cloud storage system, each cloud controller notifies the other cloud controllers. Each cloud controller then downloads these incremental metadata snapshots from the cloud storage system, updates its local metadata based on the downloaded metadata, and then deletes the downloaded incremental metadata snapshot. Other cloud controllers can then use the new metadata to access the new data stored in the incremental data snapshot (in the cloud storage system). In some scenarios, a cloud controller may also receive metadata (and data) directly from other cloud controllers. As mentioned previously, owning cloud controllers maintain data consistency by arbitrating multiple attempts to simultaneously write a given file. Cloud controllers can request updates directly from other cloud controllers if the data in the cloud storage system (or in a requesting cloud controller) is not up-to-date. In this scenario, the cloud controller storing the most recent copy of a file will determine the incremental difference between what the requesting controller has and the up-to-date version, and send the incremental difference directly to the requesting cloud controller.

Stored snapshots can be used for a range of features, including, but not limited to: “rolling-back” the filesystem to an earlier date; recovering accidentally deleted files; and archiving data from the distributed filesystem. The data structures used to store snapshots facilitate storing periodic snapshots compactly. For instance, a cloud controller may store hourly snapshots that capture the full state of the system at each hour interval; instead of storing each snapshot as a separate entity, however, these data structures may store incremental differences in a manner that is more compact but

31

still allows each hourly snapshot to be accessed as needed. For example, in this scenario the incremental difference would reference the full set of changed data and metadata for all cloud controllers over the specified time interval (as opposed to the incremental metadata and incremental data snapshots, which only specify the set of changed data and metadata for one cloud controller). When using such an organization, accessing a desired snapshot for a previous time may involve creating a copy of an even earlier snapshot, and then applying one or more incremental differences for snapshots subsequent to the earlier snapshot to generate the desired snapshot.

A cloud controller may also periodically create “full” snapshots (e.g., complete, distinct copies of the metadata for the system at a given time that can be accessed without having to apply incremental differences). In summary, in this disclosure:

1. the term “snapshot” refers to a complete set of references for the distributed filesystem at a given time;
2. a subsequent snapshot may be accessed by starting with an earlier snapshot and applying a set of incremental differences;
3. the term “incremental metadata snapshot” refers to a set of incremental differences that identify a set of metadata that has changed on a specific cloud controller since its previous snapshot;
4. the term “incremental data snapshot” refers to a set of incremental differences that include a set of data that has changed on a specific cloud controller since its previous snapshot; and
5. the term “full snapshot” refers to a specific type of snapshot that can be accessed without having to apply any incremental differences between snapshots.

FIGS. 16A-16B illustrate an exemplary process for archiving data in a distributed filesystem. FIG. 16A illustrates a set of snapshots that are taken for the distributed filesystem over time. For instance, a designated cloud controller may be configured to write a snapshot on an hourly basis. Note that storing a large number of snapshots may consume substantial storage space (even when using delta encoding to store incremental differences); hence the system may be configured to retire some of the snapshots for older time intervals. For example, the system may (as illustrated in FIG. 16A) store hourly snapshots for a recent set of days, but then only maintain a select set of daily snapshots for recent weeks, a select set of weekly snapshots for recent months, and a select set of monthly snapshots for past years.

During an archival operation, a cloud controller analyzes file metadata changes for a time interval between two snapshots (e.g., snapshots 1600 and 1602) to determine files that existed and were deleted in the time interval 1604 but are still actively stored in cloud files in a cloud storage system. For example, cloud controller 1606 may be able to determine from snapshots 1600 and 1602 (and/or the intervening and surrounding snapshots) that a given file that existed (and/or was created) in snapshot 1600 and deleted in snapshot 1602 is still stored in a cloud file in a non-archival cloud storage system. If this deletion occurred sufficiently far in the past (e.g., more than 30 days ago), the cloud controller determines that the active lifespan of this file is over, and that any data for the file that is stored in cloud files no longer need to be kept in active storage.

FIG. 16B illustrates how cloud controller 1606, after determining from past snapshots that all of the blocks in cloud file 1608 were previously deleted, moves cloud file 1608 from cloud storage system 1610 to archival cloud storage system 1612. As in FIG. 14, this transfer may involve either cloud

32

controller 1606 as an intermediary (operation 1 in FIG. 16B) or a direct transfer from cloud storage system 1610 to archival cloud storage system 1612 (operation 2 in FIG. 16B).

The archival operation illustrated in FIGS. 16A-16B describes using data gleaned from previous snapshots to determine data that can be archived. In some embodiments, cloud controllers can also actively track deletes to determine archival candidates. For instance, each cloud controller can keep track of data blocks in the cloud files it previously uploaded to the cloud storage system (e.g., by maintaining a bitmap for each cloud file), and then mark deleted blocks whenever a file is deleted. When all of the blocks in a cloud file are marked as being deleted, the owning cloud controller can move the cloud file to the archival cloud storage system. Alternatively, the cloud controller may instead add the cloud file to a special delete list for a specified interval (e.g., 30 days) to ensure that the data still remains temporarily available and quickly recoverable for that interval in case of an accidental delete or other similar issue; if no client or cloud controller accesses the cloud file during this interval, the cloud file is then moved to the archival cloud storage system. One benefit of techniques that track deletes is that they can detect and archive files that were created and deleted between snapshots (e.g., depending on the time granularity between snapshots). In some embodiments, snapshot-comparison and delete-tracking approaches can be combined into a hybrid technique that more accurately archives unneeded data.

Note that in some embodiments a cloud file typically is only archived if all of the blocks of the cloud file have been deleted; if any of the blocks have not been deleted, the cloud file remains in non-archival cloud storage. In alternative embodiments, individual blocks in a cloud file may be archived, and cloud files can be shrunk to conserve cloud storage space. For instance, if a subset of the blocks in a cloud file are moved to an archival cloud storage system, the portion of the cloud file remaining in the primary cloud storage system may need to be reconstructed to preserve the remaining blocks. For example, filling the archived blocks of a cloud file with zeros and then compressing the cloud file effectively reduces the amount of storage space needed for the empty blocks without changing block indexing and accesses for the cloud file. In a specific (small) example, for an exemplary cloud file containing blocks “ABC”, a cloud controller archiving block B could generate a modified cloud file “A0C” for the primary cloud storage system and “0B0” for the archival cloud storage system (where ‘0’ represents a block filled with zeros). If block ‘C’ were later archived, the blocks could be updated to become “A00” and “0BC”, respectively.

In some embodiments, a storage administrator manages and/or initiates archival operations. Alternatively, such operations may be scheduled at regular time intervals. Note that some archival techniques (e.g., the snapshot-comparison technique) can be initiated on a single cloud controller that has access to the collected snapshot data. Furthermore, archival operations may be initiated as low-priority background jobs (in comparison with client accesses) and/or on a cloud controller during a timeframe that the cloud controller typically has low load. Alternatively, such operations may be initiated on a backup cloud controller if available, especially if backup cloud controllers are configured to not receive client requests. Archival transfers can also be timed to occur when network bandwidth is cheaper.

In some embodiments, a distributed filesystem may include additional tables (or other structures) that facilitate accessing archived data; note that reading data from an archival cloud storage system may involve special access techniques and/or administrator assistance. After ensuring that a

cloud file has successfully been copied to an archival storage system, a cloud controller can delete that cloud file from the originating (non-archival) cloud storage system. At this point, the cloud controller may generate an incremental metadata snapshot to indicate that the metadata stored in previous snapshots can no longer be used to access the deleted data that was previously stored in the (now deleted) cloud files. However, a different set of tracking structures will need to be updated to reflect that the newly archived data in the archival cloud storage system can still be accessed if needed. For instance, archived data may be tracked in a separate set of metadata, and indexed by date such that an administrator can selectively recover and access archived data within a desired timeframe (e.g., data that is approximately one year old, data that existed during a certain timeframe, or data that was deleted between two specified dates) as needed.

Note that recovering archived data may also involve recovering archived snapshots. Because only a subset of snapshots are kept over time, a cloud controller performing an archival operation may also write an old snapshot of the distributed filesystem to the archival cloud storage provider; these old snapshots can be re-populated into the cloud controllers at a later point if needed to access cloud file data that is restored from archives. If the archived snapshot is written together with the deleted data that it references, a recovery operation may recover both simultaneously. For example, the index of archived data may indicate the deletion timeframe, original lifespan, data size, format, and identifier for each archival record, thereby facilitating archival recovery.

Note also that, in some embodiments, archival operations can also be opportunities to defragment and/or reorganize data. As described previously, cloud file size is often chosen to balance download throughput and latency; for data being archived, these are no longer issues, and hence cloud file constraints may be eliminated. For instance, file data distributed across multiple cloud files may be re-written into contiguous files (that can be recovered more easily in a single access) regardless of size during archiving. In such implementations, an index of archived files may simply store filenames, relevant metadata (e.g., creation and deletion dates, originating cloud controller, size, etc.), and a reference identifier to access the data from the archival cloud storage system.

In some embodiments, cloud controllers can also serve as intermediaries to an archival cloud storage provider for backup purposes. For instance, cloud controllers may implement a virtual tape library (VTL) interface that allows clients to perform backups as if the cloud controller were a tape backup device, with the data instead being written to the archival cloud storage provider. These techniques allow existing client backup infrastructure to be used transparently while adding additional capabilities (e.g., offsite storage in the archival cloud storage provider, redundant simultaneous writes to multiple archival cloud storage providers, etc.). Note that the high read latency of archival cloud storage providers is comparable to that of typical tape backup systems, and hence not a drawback in such usage scenarios.

FIG. 17 presents a flow chart that illustrates the process of archiving data for a distributed filesystem. Two or more cloud controllers collectively manage distributed filesystem data that is stored in one or more cloud storage systems (operation 1700); the cloud controllers cache and ensure data consistency for the stored data. Whenever each cloud controller receives new data from a client, it creates an incremental metadata snapshot for the new data that is propagated to the other cloud controllers and an incremental data snapshot containing the new data that is sent to a cloud storage system.

During operation, a cloud controller determines that a cloud file in a previously stored data snapshot is no longer being actively referenced in the distributed filesystem (operation 1710). For instance, a cloud file from a previous data snapshot 5 may no longer be referenced if all of the blocks it contains have been deleted some time ago. The cloud controller transfers this cloud file from the (first) cloud storage system to an archival cloud storage system (operation 1720), thereby reducing storage costs while preserving the data in the cloud 10 file in case it is ever needed again.

Accessing Cloud Controllers as Peer Caches

Accessing cloud files involves some overhead—a cloud controller seeking to download a target file that is not locally cached needs to: (1) contact a cloud storage system hosting one or more cloud files that contain the target file's data blocks; (2) download some or all of the target cloud files; and (3) if the cloud files are encrypted, decrypt the portions of the target cloud files that precede and include the target data blocks. Delaying factors may include the distance and net- 15 work latency between the requesting cloud controller and the cloud storage system, the load and delay in the cloud storage system, the size of cloud files, and the location of the target block in the cloud file. Furthermore, accessing the cloud storage system may be associated with an access cost and/or network bandwidth cost. Hence, while a requesting cloud controller will typically eventually get the file from the cloud storage system, accessing the data blocks for the target file may involve considerable delay. If another (peer) cloud controller in the distributed filesystem is already caching desired 20 data blocks, requesting and transferring cached data blocks from a peer cloud controller potentially facilitates accessing the desired file data more quickly (and cheaply).

In some embodiments, cloud controllers can query one or more of the other cloud controllers of the distributed filesystem to locate and access a needed data block that is already being cached by a peer cloud controller. Each individual cloud controller strives to determine the working set of its clients and locally cache a set of data that reduces the average client access latency. However, cloud controllers have limited 30 storage capacity, and hence can only cache a limited amount of data. Inter-cloud-controller cache-sharing techniques can effectively increase the size of a cached working set, thereby potentially reducing average file access latencies as well as the number of requests that are sent to the cloud storage system. Furthermore, such techniques transfer only the needed data blocks; in contrast, accessing cloud files from the cloud storage system to access the same set of data blocks potentially transfers a larger amount of data (due to the size and organization of cloud files).

However, attempts to leverage the caches of other cloud controllers can also introduce additional overhead and trade-offs. As described previously, a cloud controller writes new data received from a client to a cloud storage system. The cloud controller will typically continue to cache this data 40 locally while it is actively used by its clients; data that is no longer actively used is eventually pushed out of the local cache by other frequently and/or recently accessed data, but can be re-loaded if needed from the cloud storage provider. Thus, while it is possible that other cloud controllers may cache data that a requesting cloud controller needs, determining which cloud controllers are likely to be currently caching a needed file block is difficult. One option is to query all of the cloud controllers in the distributed filesystem. Unfortunately, while broadcasting requests to all cloud controllers increases 45 the probability of finding a cached block, processing and/or responding to such requests may substantially increase each cloud controller's load. As a result, broadcast techniques

become increasingly inefficient as the number of cloud controllers in the distributed filesystem grows. Furthermore, cloud controllers that are distributed across a large internal network or wide-area network are typically not be in the same broadcast domain; hence, broadcasts will require a large number of point-to-point messages. Such messages can increase network load, potentially to the point of slowing down accesses to the cloud storage system. Directory-based techniques that track the location and status of cached data blocks provide an alternative to broadcast-based techniques, but also require storage space and substantial message update traffic to keep the cache directories up-to-date.

In some embodiments, cloud controllers are configured to query a limited subset of peer cloud controllers in the distributed system for cached data blocks. For instance, a cloud controller may be configured to only send such “peer cache requests” to: one or more co-located cloud controllers; the cloud controller that owns (e.g., created) the cloud file containing a needed data block; a backup cloud controller; one or more cloud controllers that are determined to be in close network proximity to the requesting cloud controller; and/or one or more cloud controllers that are specified using a locality policy.

In some embodiments, a cloud controller is configured to query the caches of one or more peer cloud controllers for a data block in parallel with sending a request to the cloud storage system for the cloud file containing the data block. If a target peer cloud controller is indeed caching the data block, it can send the cached data block to the requesting cloud controller, which can then cancel and/or interrupt the transfer of the cloud file from the cloud storage system (e.g., depending on whether the cloud storage system is already transferring the cloud file or is still processing the request). If the peer cloud controller indicates that it is not caching the data block (or does not respond at all), the requesting cloud controller still receives the cloud file from the cloud storage system, and caches the requested (and now received) data block. Performing these two operations simultaneously: (1) ensures that peer cache requests do not increase the worst-case latency for a data block access; (2) adds only a small number of additional network messages and cloud controller look-ups (that are proportional to the number of peer cloud controllers that are queried); and (3) in the best case facilitates transferring needed data blocks to a requesting cloud controller more quickly than would be possible from the cloud storage system, thereby allowing the requesting cloud controller to quickly present the requested data blocks to a requesting client and cancel the parallel request to the cloud storage system before part (or sometimes even any) of the cloud file has been transferred.

FIG. 18 illustrates a cloud controller 1800 that issues peer cache requests to one or more peer cloud controllers. In one scenario, upon receiving a request for a data block of a file from client 1802, cloud controller 1800 uses the stored metadata for the distributed filesystem to determine that cloud controller 1804 is the owner of the cloud file containing the data block, and then sends a peer cache request to cloud controller 1804. More specifically, cloud controller 1800: (1) uses the stored metadata to determine the CVA for the data block; (2) determines that the data block is not currently locally cached in cloud controller 1800; (3) identifies cloud controller 1804 as the originating cloud controller for the data block based on the unique CCID in the CVA; and (4) sends a peer cache request for the data block to cloud controller 1804 (operation 2 in FIG. 18). Cloud controller 1804 is targeted for the peer cache request based on temporal and spatial locality; as the originator and a previous catcher of the data block, cloud

controller 1804 is more likely to be currently caching the data block, and thus is a good candidate for the request. As described above, cloud controller 1800 can send the peer cache request to cloud controller 1804 (operation 2) in parallel with sending a request for the cloud file referred to by the CVA to cloud storage system 302 (operation 1 in FIG. 18).

In a second scenario for FIG. 18, the distributed filesystem also includes a backup cloud controller 1806, and cloud controller 1800 instead sends a peer cache request for the data block to backup cloud controller 1806 (operation 3 in FIG. 18). As described above, backup cloud controllers may be configured to store the most frequently accessed and/or recently created data in the distributed filesystem; in this role, backup cloud controllers are likely to be caching the most useful and most-frequently-requested data blocks in the system, and are therefore logical targets for peer cache requests. Furthermore, a backup cloud controller that is not currently associated with any active clients is likely to be lightly loaded, and hence able to handle and quickly respond to peer cache requests. As in the first scenario, the peer cache request to backup cloud controller 1806 (operation 3) may be issued in parallel with the request to the cloud storage system 302 (operation 1). Alternatively, operations 1 and 2 may both be performed in parallel to operation 3.

In a third scenario for FIG. 18, cloud controllers 1800 and 1804 may be co-located at a single geographic location and collaborate to split the working set of clients at that location. For instance, two or more cloud controllers may be in very close network proximity and configured to query each other using peer cache requests before sending requests for cloud files to a cloud storage system. In such a collaborative environment, a set of tightly bound cloud controllers may operate (and segment a cache space) more efficiently if they are configured to track what their peer cloud controllers are caching (e.g., constantly update one another with the specific set of data blocks that each is caching).

In some embodiments, cloud controllers may determine a target cloud controller for a peer cache request based on a range of criteria. The preceding exemplary scenarios determined targets for peer cache requests based on ownership, proximity, and/or a specific role (e.g., the role of the backup cloud controller), but the disclosed techniques are in no way limited by these examples, and peer cache requests may be routed or partitioned based on other factors. For instance, cloud controllers may periodically calculate and share both the network latency and bandwidth between cloud controllers as well as cloud controller load (e.g., how many clients are connected to a given cloud controller, and/or how many current requests are currently queued for disk I/O), and use such information when determining the appropriateness and targets for peer cache requests. For example, a requesting cloud controller may decide not to send a peer cache request if a target cloud controller’s load is above a specified threshold, or only send high-priority requests in such instances.

In some embodiments, cloud controllers may be configured to operate in different modes during different times of the day. For instance, a cloud controller may be configured to cache data that is being actively accessed by a set of local clients during a particular time period (e.g., daytime working hours). However, when these clients are idle (e.g., during the night), this cloud controller may be configured to replace some or all of this cached data with data that is being accessed in a different time zone. More specifically, instead of being idle, this cloud controller can be configured to load a different set of data and serve this data to other active cloud controllers via peer cache requests. Such configurations may provide substantial improvements to the efficiency and average

response time of the other cloud controllers. For example, depending on the situation, retrieving a small set of data blocks from a distant cloud controller may still be faster than downloading an entire cloud file from a closer cloud storage system. Such configurations may be specified and customized for individual cloud controllers using locality policies (as described in following sections).

Note that accessing other cloud controllers as peer caches does not involve having to maintain or check coherency for data blocks. The characteristics of the transactional filesystem ensure that each data block is unique and read-only (e.g., new data is written to new blocks and new cloud files). Thus, requesting cloud controllers only need to be concerned with retrieving needed data blocks as quickly as possible, and do not need to perform checks to determine whether another cloud controller and/or client have modified the contents of the requested data block. Note also that peer cache requests are typically constant-time requests that are sent directly to target cloud controllers, and not multi-hop queries.

FIG. 19 presents a flow chart that illustrates the process of accessing cached data from a peer cloud controller in a distributed filesystem. Two or more cloud controllers collectively manage the distributed filesystem data that is stored in one or more cloud storage systems (operation 1900); the cloud controllers cache and ensure data consistency for the stored data. Whenever each cloud controller receives new data from a client, it sends an incremental metadata snapshot for the new data to the other cloud controllers and an incremental data snapshot containing the new data to a cloud storage system. During operation, a cloud controller receives a request from a client for a data block of a file stored in the distributed filesystem (operation 1910). Upon determining that the requested data block is not currently cached in the cloud controller (operation 1920), the cloud controller sends a peer cache request for the requested data block to a peer cloud controller in the distributed filesystem (operation 1930).

Global Deduplication

In some embodiments, cloud controllers use data deduplication techniques to reduce the amount of duplicate data that is stored in the distributed filesystem. A given data block may be stored multiple times in the distributed filesystem; for instance, users may backup or otherwise keep multiple copies of the same file, or may send copies of a file to other users in their organization. Over time, such duplication issues can waste a substantial amount of storage space, as the number of duplicate data blocks that are stored grows. Data duplication techniques involve calculating and tracking hash values for previously written data blocks, and comparing the hash values for newly-written data blocks against previous hash values to determine if the new data blocks have been previously stored in the distributed filesystem (and, if so, referencing the existing data block instead of writing a new, additional data block). Supporting deduplication techniques across a distributed filesystem involves sharing and distributing snapshots of deduplication information between the two or more cloud controllers that manage the data in the distributed filesystem.

Locality Policies

In some embodiments, a set of locality policies specify how data in the distributed filesystem should be managed. Such locality policies may define and/or control how data is managed at different levels of scope. For instance, locality policies may be defined at a global level (e.g., for the entire distributed filesystem) as well as at a range of finer granularities (e.g., on a per-cloud-controller, per-filesystem, per-user, per-file-type, or per-directory basis). The ability to define arbitrary locality policies that span a range of granularities

allows the behavior of the distributed filesystem to be customized to meet the needs of an organization, specific users, and/or specific applications or data sets.

In some embodiments, a locality policy can specify that one or more files should be “pinned” in the cache(s) of one or more cloud controllers. For instance, a locality policy may specify that a specific set of files that are being actively collaborated on by users throughout an organization should be pinned into the cache of every cloud controller. For example, this configuration may be achieved by specifying a locality policy that identifies one or more directories whose contents (and sub-contents) should be pinned in all cloud controllers; users can explicitly ensure that a file will be immediately distributed to and actively cached in all of the cloud controllers by storing it in such directories. In another similar example, the home directory of an important user (e.g., a chief executive officer) who frequently travels throughout an organization may be pinned so that that person can access their data with the fastest possible access times no matter where they are currently located. Alternatively, a project directory for a single site in an organization may be pinned in only the cloud controller for that site, to ensure that the data is quickly available locally (e.g., even if there is a network failure/partition between the cloud controller and the cloud storage system storing the cloud file containing that data). Note that in such situations part of the cache is fixed (e.g., may not follow normal cache replacement policies), while the remaining portions of the cache may be managed using traditional cache policies (e.g., “best effort” cache replacement policies).

In some embodiments, a locality policy can specify storage behavior for different classes of files. For instance, image and video files typically consume a substantial amount of space, and are often only viewed for a brief initial time window. Hence, a locality policy may specify that image and video files (e.g., files with “.jpg” and “.avi” extensions) should be considered low-priority (e.g., in comparison with other project file types), and should immediately be stored in a second-level (e.g., cheaper) cloud storage system instead of in a costlier high-speed, highly-replicated cloud storage system that is used for high-priority files. Conversely, other file types may be identified as being high-priority, and as a result be pinned in cloud controllers and/or stored (and kept) in the first-level cloud storage system even if they are infrequently accessed. Note that while an initial user accessing low-priority files may experience a slightly higher latency as the files are loaded from the second-level cloud storage system, subsequent users accessing these files fairly soon thereafter from the same location (e.g., before they are flushed out of the cloud controller’s cache) will be able to access the files quickly from the cache of the cloud controller.

In some embodiments, a locality policy may include time-based specifications and/or time-outs. For instance, a certain project directory may be specified to only be pinned for the lifetime of a project, or a training video may be pinned for an initial period (e.g., 15 days), and then subject to normal cache replacement policies. For example, an alternative locality policy for image files may specify that image files should be kept in a first-level cloud storage system for 30 days (or 30 days after the last access), and then moved to a second-level cloud storage system.

In some embodiments, locality policies can specify the synchronization and management of metadata and data. For example:

a locality policy for a specific cloud controller may be configured such that the cloud controller, upon receiving an incremental metadata snapshot, “warms its cache”

(e.g., preemptively downloads some of the data referenced in the incremental metadata snapshot from a cloud storage system); for instance, the cloud controller may be configured to preemptively download and cache any files created by a certain user or related to a certain project.

a locality policy may be configured to manage how data is packaged into cloud files; for instance, a locality policy may ensure that files of a similar type (e.g., image files) are packed into the same cloud file, so that the cloud file can be managed uniformly (e.g., all images files can be easily sent to a cloud storage system that is specified for image files).

a locality policy can specify a mapping of data to cloud storage providers and/or cloud storage systems; for instance, a locality policy may specify when data should be migrated between cloud storage systems, or that backup data should immediately be sent to an archival cloud storage system.

a locality policy may specify whether a cloud controller should send peer cache requests, and if so, may specify one or more peer cloud controllers that should be targeted by peer cache requests; for instance, the locality policy may specify a priority list for targeted peer cloud controllers, and/or may identify specific target peer cloud controllers based on a type of file, owner, project, etc.

a locality policy may specify when global deduplication techniques should be used; for instance, the locality policy may specify that deduplication should be disabled when a cloud controller is writing encrypted files (which frequently do not benefit from deduplication efforts).

In summary, embodiments of the present invention facilitate storing and accessing data in a distributed filesystem. A set of distributed cloud controllers manage data stored in a cloud-based storage system to provide a high-capacity, high-reliability storage system that ensures data consistency. These cloud controllers cache the set of data that is being used by their respective clients, store updates in cloud files on the cloud storage system, and forward updates to each other via incremental snapshots. The data capacity of the system can be easily extended as needed by leasing additional space for the cloud storage system. Hence, the disclosed embodiments present an abstraction of one global, extensible filesystem while preserving the abstraction of high-speed local data access.

Computing Environment

In some embodiments of the present invention, techniques for managing and/or accessing a distributed filesystem can be incorporated into a wide range of computing devices in a computing environment. For example, FIG. 9 illustrates a computing environment 900 in accordance with an embodiment of the present invention. Computing environment 900 includes a number of computer systems, which can generally include any type of computer system based on a microprocessor, a mainframe computer, a digital signal processor, a portable computing device, a personal organizer, a device controller, or a computational engine within an appliance. More specifically, referring to FIG. 9, computing environment 900 includes clients 910-912, users 920 and 921, servers 930-950, network 960, database 970, devices 980, appliance 990, and cloud-based storage system 995.

Clients 910-912 can include any node on a network that includes computational capability and includes a mechanism for communicating across the network. Additionally, clients 910-912 may comprise a tier in an n-tier application archi-

ecture, wherein clients 910-912 perform as servers (servicing requests from lower tiers or users), and wherein clients 910-912 perform as clients (forwarding the requests to a higher tier).

Similarly, servers 930-950 can generally include any node on a network including a mechanism for servicing requests from a client for computational and/or data storage resources. Servers 930-950 can participate in an advanced computing cluster, or can act as stand-alone servers. For instance, computing environment 900 can include a large number of compute nodes that are organized into a computing cluster and/or server farm. In one embodiment of the present invention, server 940 is an online “hot spare” of server 950.

Users 920 and 921 can include: an individual; a group of individuals; an organization; a group of organizations; a computing system; a group of computing systems; or any other entity that can interact with computing environment 900.

Network 960 can include any type of wired or wireless communication channel capable of coupling together computing nodes. This includes, but is not limited to, a local area network, a wide area network, or a combination of networks. In one embodiment of the present invention, network 960 includes the Internet. In some embodiments of the present invention, network 960 includes phone and cellular phone networks.

Database 970 can include any type of system for storing data in non-volatile storage. This includes, but is not limited to, systems based upon magnetic, optical, or magneto-optical storage devices, as well as storage devices based on flash memory and/or battery-backed up memory. Note that database 970 can be coupled: to a server (such as server 950), to a client, or directly to a network. In some embodiments of the present invention, database 970 is used to store information that may later be stored in unused bits of a memory pointer. Alternatively, other entities in computing environment 900 (e.g., servers 930-950) may also store such data.

Devices 980 can include any type of electronic device that can be coupled to a client, such as client 912. This includes, but is not limited to, cell phones, personal digital assistants (PDAs), smartphones, personal music players (such as MP3 players), gaming systems, digital cameras, portable storage media, or any other device that can be coupled to the client. Note that, in some embodiments of the present invention, devices 980 can be coupled directly to network 960 and can function in the same manner as clients 910-912.

Appliance 990 can include any type of appliance that can be coupled to network 960. This includes, but is not limited to, routers, switches, load balancers, network accelerators, and specialty processors. Appliance 990 may act as a gateway, a proxy, or a translator between server 940 and network 960.

Cloud-based storage system 995 can include any type of networked storage devices (e.g., a federation of homogeneous or heterogeneous storage devices) that together provide data storage capabilities to one or more clients.

Note that different embodiments of the present invention may use different system configurations, and are not limited to the system configuration illustrated in computing environment 900. In general, any device that includes computational and storage capabilities may incorporate elements of the present invention.

FIG. 10 illustrates a computing device 1000 that includes a processor 1002 and a storage mechanism 1004. Computing device 1000 also includes a receiving mechanism 1006 and a storage management mechanism 1008.

In some embodiments, computing device 1000 uses receiving mechanism 1006, storage management mechanism 1008, and storage mechanism 1004 to manage data in a distributed

41

filesystem. For instance, storage mechanism **1004** can store metadata for a distributed filesystem, and computing device **1000** can use receiving mechanism **1006** to receive a request to access a data block for a file. Program instructions executing on processor **1002** can traverse the stored metadata to identify a metadata entry that is associated with the data block. Storage management mechanism **1008** can use this metadata entry to download a cloud file containing the data block from a cloud storage system.

In some embodiments of the present invention, some or all aspects of receiving mechanism **1006**, storage management mechanism **1008**, and/or a filesystem device driver can be implemented as dedicated hardware modules in computing device **1000**. These hardware modules can include, but are not limited to, processor chips, application-specific integrated circuit (ASIC) chips, field-programmable gate arrays (FPGAs), memory chips, and other programmable-logic devices now known or later developed.

Processor **1002** can include one or more specialized circuits for performing the operations of the mechanisms. Alternatively, some or all of the operations of receiving mechanism **1006**, storage management mechanism **1008**, and/or a filesystem device driver may be performed using general-purpose circuits in processor **1002** that are configured using processor instructions. Thus, while FIG. 10 illustrates receiving mechanism **1006** and/or storage management mechanism **1008** as being external to processor **1002**, in alternative embodiments some or all of these mechanisms can be internal to processor **1002**.

In these embodiments, when the external hardware modules are activated, the hardware modules perform the methods and processes included within the hardware modules. For example, in some embodiments of the present invention, the hardware module includes one or more dedicated circuits for performing the operations described above. As another example, in some embodiments of the present invention, the hardware module is a general-purpose computational circuit (e.g., a microprocessor or an ASIC), and when the hardware module is activated, the hardware module executes program code (e.g., BIOS, firmware, etc.) that configures the general-purpose circuits to perform the operations described above.

The foregoing descriptions of various embodiments have been presented only for purposes of illustration and description. They are not intended to be exhaustive or to limit the present invention to the forms disclosed. Accordingly, many modifications and variations will be apparent to practitioners skilled in the art. Additionally, the above disclosure is not intended to limit the present invention. The scope of the present invention is defined by the appended claims.

What is claimed is:

1. A computer-implemented method for accessing cached data from a peer cloud controller in a distributed filesystem, the method comprising:

collectively managing the data of the distributed filesystem

using two or more cloud controllers, wherein collectively managing the data comprises:

collectively presenting a unified namespace for the distributed filesystem to the clients of the distributed filesystem via the two or more cloud controllers, wherein the clients can only access the distributed filesystem via the cloud controllers, wherein the file data for the distributed filesystem is stored in a remote cloud storage system using fixed-size cloud files, wherein each cloud controller caches a subset of the file data from the remote cloud storage system that is being actively accessed by that cloud controller's respective clients, wherein all new file data received

42

by each cloud controller from its clients is written to the remote cloud storage system via the receiving cloud controller;

maintaining at each cloud controller a copy of the complete metadata for all of the files stored in the distributed filesystem, wherein each cloud controller communicates any changes to the metadata for the distributed filesystem to the other cloud controllers for the distributed filesystem to ensure that the clients of the distributed filesystem share a consistent view of each file in the distributed filesystem;

upon receiving in a cloud controller new data from a client, storing the new file data for the distributed filesystem as a cloud file in the remote cloud storage system, wherein the cloud file is sent from the cloud controller to the remote cloud storage system as part of an incremental data snapshot; and

upon receiving confirmation that the cloud file has been successfully stored in the remote cloud storage system, sending from the cloud controller an incremental metadata snapshot that includes new metadata for the distributed filesystem that describes the new data, wherein the incremental metadata snapshot is received by the other cloud controllers of the distributed filesystem;

receiving at the cloud controller a request from the client for a data block for a file in the distributed filesystem; determining that the requested data block is not currently cached in the cloud controller;

determining that the peer cloud controller in the distributed filesystem is likely to cache the requested data block; and

sending a peer cache request for the requested data block from the cloud controller to the peer cloud controller.

2. The computer-implemented method of claim 1, wherein sending the peer cache request further comprises sending the peer cache request in parallel with sending a request to the remote cloud storage system for a cloud file that contains the requested data block.

3. The computer-implemented method of claim 2, wherein if the peer cloud controller is currently caching the requested data block, the method further comprises:

sending the requested data block from the peer cloud controller to the requesting cloud controller; and

sending a cancellation request from the cloud controller to the remote cloud storage system to cancel the request for the cloud file.

4. The computer-implemented method of claim 2, wherein if the peer cloud controller is not currently caching the requested data block, the method further comprises:

receiving the cloud file from the remote cloud storage system; and

extracting the requested data block from the received cloud file.

5. The computer-implemented method of claim 2, wherein sending the peer cache request further comprises:

determining a cloud virtual address (CVA) associated with the cloud file in the remote cloud storage system that contains the requested data block; and

determining from the CVA a unique cloud controller identifier associated with the cloud file, wherein the unique cloud controller identifier identifies an originating cloud controller that originally uploaded the cloud file to the remote cloud storage system; and

wherein if the originating cloud controller is different from the requesting cloud controller, sending the peer cache request to the originating cloud controller.

43

6. The computer-implemented method of claim 2, wherein sending the peer cache request further comprises:

determining the presence of a backup cloud controller for the distributed filesystem, wherein the backup cloud controller is configured to determine and cache a set of data that is most frequently accessed by the clients of the distributed filesystem; and
sending the peer cache request to the backup cloud controller.

7. The computer-implemented method of claim 2,
wherein the requesting cloud controller is co-located with one or more other cloud controllers for the distributed filesystem;
wherein the co-located cloud controllers are configured to collaboratively cache data for a set of clients by tracking the set of data that is being cached by each co-located cloud controller; and
wherein sending the peer cache request further comprises sending the peer cache request to a co-located cloud controller that is caching the requested data block.

8. The computer-implemented method of claim 2, wherein sending the peer cache request further comprises determining the peer cloud controller using a locality policy for the requesting cloud controller that specifies the peer cloud controller.

9. The computer-implemented method of claim 2, wherein sending the peer cache request further comprises determining the peer cloud controller by:

determining the network latency and network bandwidth between the cloud controller and the other cloud controllers of the distributed filesystem; and
sending the peer cache request to one or more cloud controllers that are determined to be in close network proximity to the requesting cloud controller.

10. The computer-implemented method of claim 2, wherein sending the peer cache request further comprises determining the peer cloud controller by:

determining the load for the other cloud controllers of the distributed filesystem; and
sending the peer cache request to one or more cloud controllers that are determined to have a low load.

11. The computer-implemented method of claim 2,
wherein the peer cloud controller is configured to cache data for an associated set of clients during a first time interval;

wherein the peer cloud controller is configured to cache data for the requesting cloud controller for a second time interval during which the associated set of clients are likely to be idle; and

wherein sending the peer cache request further comprises sending the peer cache request to the peer cloud controller during the second time interval.

12. The computer-implemented method of claim 2, wherein sending the peer cache request further comprises sending the peer cache request to two or more cloud controllers including:

an originating cloud controller that originally uploaded the cloud file to the remote cloud storage system;
a backup cloud controller for the distributed filesystem;
one or more cloud controllers that are co-located with the requesting cloud controller;
one or more peer cloud controllers specified in a locality policy for the requesting cloud controller;
one or more peer cloud controllers that are determined to be in close network proximity to the requesting cloud controller;

44

one or more peer cloud controllers that are determined to have a low load; and
the peer cloud controller during a time interval in which the peer cloud controller is configured to be caching data on behalf of the requesting cloud controller.

13. The computer-implemented method of claim 2,
wherein the remote cloud storage system is unaware of the organization and structure of the distributed filesystem; and
wherein data stored in the distributed filesystem is indexed

using a global address space;
wherein data is stored in the remote cloud storage system as cloud files, wherein each cloud file is uniquely indexed in the global address space;

wherein a second cloud controller receiving the incremental metadata snapshot uses a metadata entry in the incremental metadata snapshot to retrieve data in the incremental data snapshot; and

wherein using the metadata entry to retrieve data in the incremental data snapshot comprises:

determining from the metadata entry that a desired data block in the incremental data snapshot is not presently stored in the second cloud controller;
using a global address stored in the metadata entry to identify a cloud file in the remote cloud storage system that includes the data block;
downloading the identified cloud file to the second cloud controller; and
using an offset stored in the metadata entry to determine the location of the data block in the cloud file.

14. A non-transitory computer-readable storage medium storing instructions that when executed by a computer cause the computer to perform a method for accessing cached data from a peer cloud controller in a distributed filesystem, the method comprising:

collectively managing the data of the distributed filesystem using two or more cloud controllers, wherein collectively managing the data comprises:

collectively presenting a unified namespace for the distributed filesystem to the clients of the distributed filesystem via the two or more cloud controllers, wherein the clients can only access the distributed filesystem via the cloud controllers, wherein the file data for the distributed filesystem is stored in a remote cloud storage system using fixed-size cloud files, wherein each cloud controller caches a subset of the file data from the remote cloud storage system that is being actively accessed by that cloud controller's respective clients, wherein all new file data received by each cloud controller from its clients is written to the remote cloud storage system via the receiving cloud controller;

maintaining at each cloud controller a copy of the complete metadata for all of the files stored in the distributed filesystem, wherein each cloud controller communicates any changes to the metadata for the distributed filesystem to the other cloud controllers for the distributed filesystem to ensure that the clients of the distributed filesystem share a consistent view of each file in the distributed filesystem;

upon receiving in a cloud controller new data from a client, storing the new file data for the distributed filesystem as a cloud file in the remote cloud storage system, wherein the cloud file is sent from the cloud controller to the remote cloud storage system as part of an incremental data snapshot; and

upon receiving confirmation that the cloud file has been successfully stored in the remote cloud storage sys-

45

tem, sending from the cloud controller an incremental metadata snapshot that includes new metadata for the distributed filesystem that describes the new data, wherein the incremental metadata snapshot is received by the other cloud controllers of the distributed filesystem; 5 receiving at the cloud controller a request from the client for a data block for a file in the distributed filesystem; determining that the requested data block is not currently cached in the cloud controller; 10 determining that the peer cloud controller in the distributed filesystem is likely to cache the requested data block; and sending a peer cache request for the requested data block 15 from the cloud controller to the peer cloud controller.

15. The non-transitory computer-readable storage medium of claim **14**, wherein sending the peer cache request further comprises sending the peer cache request in parallel with sending a request to the remote cloud storage system for a 20 cloud file that contains the data block.

16. The non-transitory computer-readable storage medium of claim **15**, wherein if the peer cloud controller is currently caching the requested data block, the method further comprises:

sending the requested data block from the peer cloud controller to the requesting cloud controller; and sending a cancellation request from the cloud controller to the remote cloud storage system to cancel the request for the cloud file.

17. The non-transitory computer-readable storage medium of claim **15**, wherein if the peer cloud controller is not currently caching the requested data block, the method further comprises:

receiving the cloud file from the remote cloud storage system; and extracting the requested data block from the received cloud file.

18. The non-transitory computer-readable storage medium of claim **15**, wherein sending the peer cache request further comprises:

determining a cloud virtual address (CVA) associated with the cloud file in the remote cloud storage system that contains the requested data block; and

determining from the CVA a unique cloud controller identifier associated with the cloud file, wherein the unique cloud controller identifier identifies an originating cloud controller that originally uploaded the cloud file to the remote cloud storage system; and

wherein if the originating cloud controller is different from the requesting cloud controller, sending the peer cache 50 request to the originating cloud controller.

19. The non-transitory computer-readable storage medium of claim **15**, wherein sending the peer cache request further comprises:

determining the presence of a backup cloud controller for the distributed filesystem, wherein the backup cloud controller is configured to determine and cache a set of data that is most frequently accessed by the clients of the distributed filesystem; and

46

sending the peer cache request to the backup cloud controller.

20. A cloud controller that accesses cached data from a peer cloud controller in a distributed filesystem, comprising: a processor;

a storage mechanism that stores metadata for the distributed filesystem; and

a storage management mechanism; wherein two or more cloud controllers collectively manage the data of the distributed filesystem, wherein collectively managing the data comprises:

collectively presenting a unified namespace for the distributed filesystem to the clients of the distributed filesystem via the two or more cloud controllers, wherein the clients can only access the distributed filesystem via the cloud controllers, wherein the file data for the distributed filesystem is stored in a remote cloud storage system using fixed-size cloud files, wherein each cloud controller caches a subset of the file data from the remote cloud storage system that is being actively accessed by that cloud controller's respective clients, wherein all new file data received by each cloud controller from its clients is written to the remote cloud storage system via the receiving cloud controller;

maintaining at each cloud controller a copy of the complete metadata for all of the files stored in the distributed filesystem, wherein each cloud controller communicates any changes to the metadata for the distributed filesystem to the other cloud controllers for the distributed filesystem to ensure that the clients of the distributed filesystem share a consistent view of each file in the distributed filesystem;

upon receiving in a cloud controller new data from a client, storing the new file data for the distributed filesystem as a cloud file in the remote cloud storage system, wherein the cloud file is sent from the cloud controller to the remote cloud storage system as part of an incremental data snapshot; and

upon receiving confirmation that the cloud file has been successfully stored in the remote cloud storage system, sending from the cloud controller an incremental metadata snapshot that includes new metadata for the distributed filesystem that describes the new data, wherein the incremental metadata snapshot is received by the other cloud controllers of the distributed filesystem;

wherein the storage management mechanism is configured to:

receive a request from the client for a data block for a file in the distributed filesystem;

determine that the requested data block is not currently cached in the cloud controller;

determine that the peer cloud controller in the distributed filesystem is likely to cache the requested data block; and

send a peer cache request for the requested data block from the cloud controller to the peer cloud controller.

* * * * *