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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
16 May 2019 (16.05.2019)

WIPO | PCT

(10) International Publication Number
WO 2019/092501 A1

(51) International Patent Classification:
G06F 12/02 (2006.01) *G06F 11/07* (2006.01)
G06F 12/16 (2006.01)

(21) International Application Number:
PCT/IB2018/001316

(22) International Filing Date:
05 October 2018 (05.10.2018)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/585,057 13 November 2017 (13.11.2017) US
16/121,508 04 September 2018 (04.09.2018) US

(71) Applicant: **WEKA.IO LTD.** [IL/IL]; Bait Shamai 10, 6701838 Tel Aviv (IL).

(72) Inventors: **PALMON, Omri**; Beit Shamai 10, 6701838 Tel Aviv (IL). **BEN-DAYAN, Maor**; Beit Shamai 10, 6701838 Tel Aviv (IL). **ZVIBEL, Kanael**; Beit Shamai 10, 6701838 Tel Aviv (IL). **ARDITTI, Kanael**; Beit Shamai 10, 6701838 Tel Aviv (IL).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH,

(54) Title: FILE OPERATIONS IN A DISTRIBUTED STORAGE SYSTEM

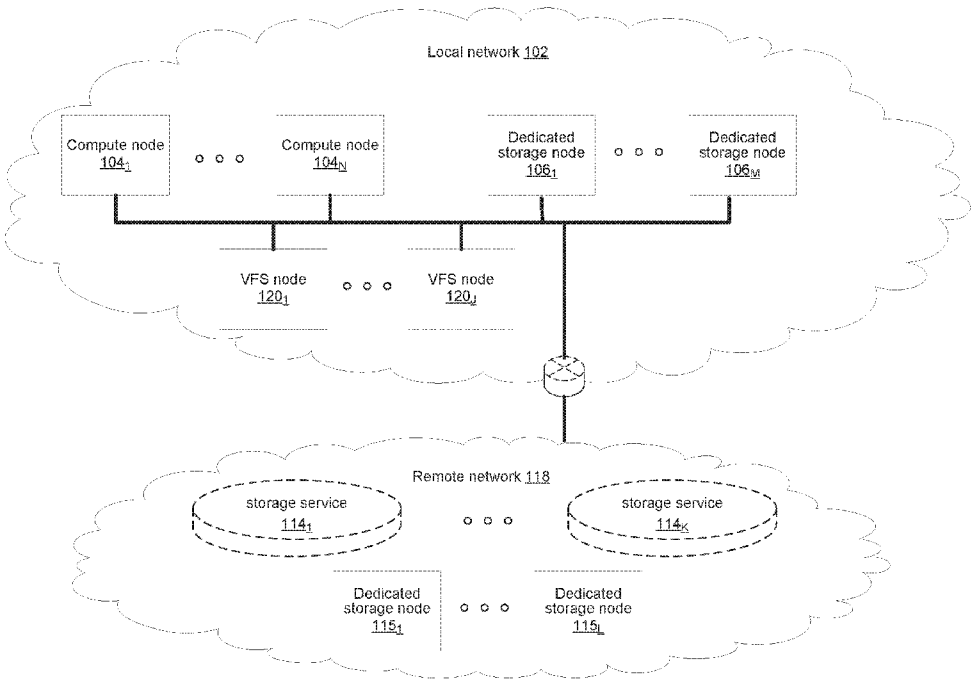

FIG. 1

(57) Abstract: A plurality of computing devices are communicatively coupled to each other via a network, and each of the plurality of computing devices is operably coupled to one or more of a plurality of storage devices. A plurality of failure resilient address spaces are distributed across the plurality of storage devices such that each of the plurality of failure resilient address spaces spans a plurality of the storage devices. The plurality of computing devices maintains metadata that maps each failure resilient address space to one of the plurality of computing devices. Each of the plurality of computing devices is operable to read from and write to a plurality of memory blocks, while maintaining an extent in metadata that maps the plurality of memory blocks to the failure resilient address space.

WO 2019/092501 A1

[Continued on next page]

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

Published:

— *with international search report (Art. 21(3))*

FILE OPERATIONS IN A DISTRIBUTED STORAGE SYSTEM

PRIORITY CLAIM

[0001] This application claims priority to United States provisional patent application 62/585,057 titled "File Operations in a Distributed Storage System" filed on November 13, 2017, and U.S. Patent Application No. 16/121,508, titled "File Operations in a Distributed Storage System" filed on September 4, 2018.

BACKGROUND

[0002] Limitations and disadvantages of conventional approaches to data storage will become apparent to one of skill in the art, through comparison of such approaches with some aspects of the present method and system set forth in the remainder of this disclosure with reference to the drawings.

INCORPORATION BY REFERENCE

[0003] United States Patent Application No. 15/243,519 titled "Distributed Erasure Coded Virtual File System" is hereby incorporated herein by reference in its entirety.

BRIEF SUMMARY

[0004] Methods and systems are provided for file operations in a distributed storage system substantially as illustrated by and/or described in connection with at least one of the figures, as set forth more completely in the claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] FIG. 1 illustrates various example configurations of a virtual file system in accordance with aspects of this disclosure.

[0006] FIG. 2 illustrates an example configuration of a virtual file system node in accordance with aspects of this disclosure.

[0007] FIG. 3 illustrates another representation of a virtual file system in accordance with an example implementation of this disclosure.

[0008] FIG. 4 illustrates an example of a flash registry with write leveling in accordance with an example implementation of this disclosure.

[0009] FIG. 5 illustrates an example of a shadow registry associated with the flash registry in FIG. 4 in accordance with an example implementation of this disclosure.

[0010] FIG. 6 illustrates an example implementation in which two distributed failure resilient address spaces reside on a plurality of solid-state storage disks.

[0011] FIG. 7 illustrates a forward error correction scheme which may be used for protecting data stored to nonvolatile memory of a virtual file system in accordance with an example implementation of this disclosure.

DETAILED DESCRIPTION

[0012] The systems in this disclosure are applicable to small clusters and can also scale to many, many thousands of nodes. An example embodiment is discussed regarding non-volatile memory (NVM), for example, flash memory that comes in the form of a solid-state drive (SSD). The NVM may be divided into 4kB "blocks" and 128MB "chunks." "Extents" may be stored in volatile memory, e.g., RAM for fast access, backed up by NVM storage as well. An extent may store pointers for blocks, e.g., 256 pointers to 1MB of data stored in blocks. In other embodiments, larger or smaller memory divisions may also be used. Metadata functionality in this disclosure may be effectively spread across many servers. For example, in cases of "hot spots" where a large load is targeted at a specific portion of the filesystem's namespace, this load can be distributed across a plurality of nodes.

[0013] FIG. 1 illustrates various example configurations of a virtual file system (VFS) in accordance with aspects of this disclosure. Shown in FIG. 1 is a local area network (LAN) 102 comprising one or more VFS nodes 120 (indexed by integers from 1 to J, for $j \geq 1$), and optionally comprising (indicated by dashed lines): one or more dedicated storage nodes 106 (indexed by integers from 1 to M, for $M \geq 1$), one or more compute nodes 104 (indexed by integers from 1 to N, for $N \geq 1$), and/or an edge router that connects the LAN 102 to a remote network 118. The remote network 118 optionally comprises one or more storage services 114 (indexed by integers from 1 to K, for $K \geq 1$), and/or one or more dedicated storage nodes 115 (indexed by integers from 1 to L, for $L \geq 1$).

[0014] Each VFS node 120_j (j an integer, where $1 \leq j \leq J$) is a networked computing device (e.g., a server, personal computer, or the like) that comprises circuitry for running VFS processes and, optionally, client processes (either directly on an

operating system of the device 104_n and/or in one or more virtual machines running in the device 104_n).

[0015] The compute nodes 104 are networked devices that may run a VFS frontend without a VFS backend. A compute node 104 may run VFS frontend by taking an SR-IOV into the NIC and consuming a complete processor core. Alternatively, the compute node 104 may run the VFS frontend by routing the networking through a Linux kernel networking stack and using kernel process scheduling, thus not having the requirement of a full core. This is useful if a user does not want to allocate a complete core for the VFS or if the networking hardware is incompatible with the VFS requirements.

[0016] FIG. 2 illustrates an example configuration of a VFS node in accordance with aspects of this disclosure. A VFS node comprises a VFS frontend 202 and driver 208, a VFS memory controller 204, a VFS backend 206, and a VFS SSD agent 214. As used in this disclosure, a "VFS process" is a process that implements one or more of: the VFS frontend 202, the VFS memory controller 204, the VFS backend 206, and the VFS SSD agent 214. Thus, in an example implementation, resources (e.g., processing and memory resources) of the VFS node may be shared among client processes and VFS processes. The processes of the VFS may be configured to demand relatively small amounts of the resources to minimize the impact on the performance of the client applications. The VFS frontend 202, the VFS memory controller 204, and/or the VFS backend 206 and/or the VFS SSD agent 214 may run on a processor of the host 201 or on a processor of the network adaptor 218. For a multi-core processor, different VFS process may run on different cores, and may run a different subset of the services. From the perspective of the client process(es) 212, the interface with the virtual file system is independent of the particular physical machine(s) on which the VFS process(es) are

running. Client processes only require driver 208 and frontend 202 to be present in order to serve them.

[0017] The VFS node may be implemented as a single tenant server (e.g., bare-metal) running directly on an operating system or as a virtual machine (VM) and/or container (e.g., a Linux container (LXC)) within a bare-metal server. The VFS may run within an LXC container as a VM environment. Thus, inside the VM, the only thing that may run is the LXC container comprising the VFS. In a classic bare-metal environment, there are user-space applications and the VFS runs in an LXC container. If the server is running other containerized applications, the VFS may run inside an LXC container that is outside the management scope of the container deployment environment (e.g. Docker).

[0018] The VFS node may be serviced by an operating system and/or a virtual machine monitor (VMM) (e.g., a hypervisor). The VMM may be used to create and run the VFS node on a host 201. Multiple cores may reside inside the single LXC container running the VFS, and the VFS may run on a single host 201 using a single Linux kernel. Therefore, a single host 201 may comprise multiple VFS frontends 202, multiple VFS memory controllers 204, multiple VFS backends 206, and/or one or more VFS drivers 208. A VFS driver 208 may run in kernel space outside the scope of the LXC container.

[0019] A single root input/output virtualization (SR-IOV) PCIe virtual function may be used to run the networking stack 210 in user space 222. SR-IOV allows the isolation of PCI Express, such that a single physical PCI Express can be shared on a virtual environment and different virtual functions may be offered to different virtual components on a single physical server machine. The I/O stack 210 enables the VFS node to bypasses the standard TCP/IP stack 220 and communicate directly with the network adapter 218. A Portable Operating System Interface for uniX (POSIX) VFS functionality may be provided through lockless queues to the VFS driver 208. SR-IOV or full PCIe physical function address may also be used to run non-volatile memory

express (NVMe) driver 214 in user space 222, thus bypassing the Linux IO stack completely. NVMe may be used to access non-volatile storage media 216 attached via a PCI Express (PCIe) bus. The non-volatile storage media 216 may be, for example, flash memory that comes in the form of a solid-state drive (SSD) or Storage Class Memory (SCM) that may come in the form of an SSD or a memory module (DIMM). Other example may include storage class memory technologies such as 3D-XPoint.

[0020] The SSD may be implemented as a networked device by coupling the physical SSD 216 with the SSD agent 214 and networking 210. Alternatively, the SSD may be implemented as a network-attached NVMe SSD 242 or 244 by using a network protocol such as NVMe-oF (NVMe over Fabrics). NVMe-oF may allow access to the NVMe device using redundant network links, thereby providing a higher level or resiliency. Network adapters 226, 228, 230 and 232 may comprise hardware acceleration for connection to the NVMe SSD 242 and 244 to transform them into networked NVMe-oF devices without the use of a server. The NVMe SSDs 242 and 244 may each comprise two physical ports, and all the data may be accessed through either of these ports.

[0021] Each client process/application 212 may run directly on an operating system or may run in a virtual machine and/or container serviced by the operating system and/or hypervisor. A client process 212 may read data from storage and/or write data to storage in the course of performing its primary function. The primary function of a client process 212, however, is not storage-related (i.e., the process is only concerned that its data is reliably stored and is retrievable when needed, and not concerned with where, when, or how the data is stored). Example applications which give rise to such processes include: email servers, web servers, office productivity applications, customer relationship management (CRM), animated video rendering, genomics calculation, chip design, software builds, and enterprise resource planning (ERP).

[0022] A client application 212 may make a system call to the kernel 224 which communicates with the VFS driver 208. The VFS driver 208 puts a corresponding request on a queue of the VFS frontend 202. If several VFS frontends exist, the driver may load balance accesses to the different frontends, making sure a single file/directory is always accessed via the same frontend. This may be done by "sharding" the frontend based on the ID of the file or directory. The VFS frontend 202 provides an interface for routing file system requests to an appropriate VFS backend based on the bucket that is responsible for that operation. The appropriate VFS backend may be on the same host or it may be on another host.

[0023] The VFS backend 206 hosts several buckets, each one of them services the file system requests that it receives and carries out tasks to otherwise manage the virtual file system (e.g., load balancing, journaling, maintaining metadata, caching, moving of data between tiers, removing stale data, correcting corrupted data, etc.)

[0024] The VFS SSD agent 214 handles interactions with a respective storage device 216. This may include, for example, translating addresses, and generating the commands that are issued to the storage device (e.g., on a SATA, SAS, PCIe, or other suitable bus). Thus, the VFS SSD agent 214 operates as an intermediary between a storage device 216 and the VFS backend 206 of the virtual file system. The SSD agent 214 could also communicate with a standard network storage device supporting a standard protocol such as NVMe-oF (NVMe over Fabrics).

[0025] FIG. 3 illustrates another representation of a virtual file system in accordance with an example implementation of this disclosure. In FIG. 3, the element 302 represents memory resources (e.g., DRAM and/or other short-term memory) and processing (e.g., x86 processor(s), ARM processor(s), NICs, ASICs, FPGAs, and/or the like) resources of various node(s) (compute, storage, and/or VFS) on which resides a virtual file system, such as described regarding FIG. 2 above. The element 308

represents the one or more physical storage devices 216 which provide the long term storage of the virtual file system.

[0026] As shown in FIG. 3, the physical storage is organized into a plurality of distributed failure resilient address spaces (DFRASs) 518. Each of which comprises a plurality of chunks 310, which in turn comprises a plurality of blocks 312. The organization of blocks 312 into chunks 310 is only a convenience in some implementations and may not be done in all implementations. Each block 312 stores committed data 316 (which may take on various states, discussed below) and/or metadata 314 that describes or references committed data 316.

[0027] The organization of the storage 308 into a plurality of DFRASs enables high performance parallel commits from many – perhaps all – of the nodes of the virtual file system (e.g., all nodes 104_1–104_N, 106_1–106_M, and 120_1–120_J of FIG. 1 may perform concurrent commits in parallel). In an example implementation, each of the nodes of the virtual file system may own a respective one or more of the plurality of DFRAS and have exclusive read/commit access to the DFRASs that it owns.

[0028] Each bucket owns a DFRAS, and thus does not need to coordinate with any other node when writing to it. Each bucket may build stripes across many different chunks on many different SSDs, thus each bucket with its DFRAS can choose what "chunk stripe" to write to currently based on many parameters, and there is no coordination required in order to do so once the chunks are allocated to that bucket. All buckets can effectively write to all SSDs without any need to coordinate.

[0029] Each DFRAS being owned and accessible by only its owner bucket that runs on a specific node allows each of the nodes of the VFS to control a portion of the storage 308 without having to coordinate with any other nodes (except during [re]assignment of the buckets holding the DFRASs during initialization or after a node failure, for example, which may be performed asynchronously to actual reads/commits to

storage 308). Thus, in such an implementation, each node may read/commit to its buckets' DFRASs independently of what the other nodes are doing, with no requirement to reach any consensus when reading and committing to storage 308. Furthermore, in the event of a failure of a particular node, the fact the particular node owns a plurality of buckets permits more intelligent and efficient redistribution of its workload to other nodes (rather the whole workload having to be assigned to a single node, which may create a "hot spot"). In this regard, in some implementations the number of buckets may be large relative to the number of nodes in the system such that any one bucket may be a relatively small load to place on another node. This permits fine grained redistribution of the load of a failed node according to the capabilities and capacity of the other nodes (e.g., nodes with more capabilities and capacity may be given a higher percentage of the failed nodes buckets).

[0030] To permit such operation, metadata may be maintained that maps each bucket to its current owning node such that reads and commits to storage 308 can be redirected to the appropriate node.

[0031] Load distribution is possible because the entire filesystem metadata space (e.g., directory, file attributes, content range in the file, etc.) can be broken (e.g., chopped or sharded) into small, uniform pieces (e.g., "shards"). For example, a large system with 30k servers could chop the metadata space into 128k or 256k shards.

[0032] Each such metadata shard may be maintained in a "bucket." Each VFS node may have responsibility over several buckets. When a bucket is serving metadata shards on a given backend, the bucket is considered "active" or the "leader" of that bucket. Typically, there are many more buckets than VFS nodes. For example, a small system with 6 nodes could have 120 buckets, and a larger system with 1,000 nodes could have 8k buckets.

[0033] Each bucket may be active on a small set of nodes, typically 5 nodes that that form a penta-group for that bucket. The cluster configuration keeps all participating nodes up-to-date regarding the penta-group assignment for each bucket.

[0034] Each penta-group monitors itself. For example, if the cluster has 10k servers, and each server has 6 buckets, each server will only need to talk with 30 different servers to maintain the status of its buckets (6 buckets will have 6 penta-groups, so 6*5 = 30). This is a much smaller number than if a centralized entity had to monitor all nodes and keep a cluster-wide state. The use of penta-groups allows performance to scale with bigger clusters, as nodes do not perform more work when the cluster size increases. This could pose a disadvantage that in a "dumb" mode a small cluster could actually generate more communication than there are physical nodes, but this disadvantage is overcome by sending just a single heartbeat between two servers with all the buckets they share (as the cluster grows this will change to just one bucket, but if you have a small 5 server cluster then it will just include all the buckets in all messages and each server will just talk with the other 4). The penta-groups may decide (i.e., reach consensus) using an algorithm that resembles the Raft consensus algorithm.

[0035] Each bucket may have a group of compute nodes that can run it. For example, five VFS nodes can run one bucket. However, only one of the nodes in the group is the controller/leader at any given moment. Further, no two buckets share the same group, for large enough clusters. If there are only 5 or 6 nodes in the cluster, most buckets may share backends. In a reasonably large cluster there many distinct node groups. For example, with 26 nodes, there are more than 64,000 ($\frac{26!}{5!*(26-5)!}$) possible five-node groups (i.e., penta-groups).

[0036] All nodes in a group know and agree (i.e., reach consensus) on which node is the actual active controller (i.e., leader) of that bucket. A node accessing the bucket may remember ("cache") the last node that was the leader for that bucket out of the (e.g.,

five) members of a group. If it accesses the bucket leader, the bucket leader performs the requested operation. If it accesses a node that is not the current leader, that node indicates the leader to "redirect" the access. If there is a timeout accessing the cached leader node, the contacting node may try a different node of the same penta-group. All the nodes in the cluster share common "configuration" of the cluster, which allows the nodes to know which server may run each bucket.

[0037] Each bucket may have a load/usage value that indicates how heavily the bucket is being used by applications running on the filesystem. For example, a server node with 11 lightly used buckets may receive another bucket of metadata to run before a server with 9 heavily used buckets, even though there will be an imbalance in the number of buckets used. Load value may be determined according to average response latencies, number of concurrently run operations, memory consumed or other metrics.

[0038] Redistribution may also occur even when a VFS node does not fail. If the system identifies that one node is busier than the others based on the tracked load metrics, the system can move (i.e., "fail over") one of its buckets to another server that is less busy. However, before actually relocating a bucket to a different host, load balancing may be achieved by diverting writes and reads. Since each write may end up on a different group of nodes, decided by the DFRAS, a node with a higher load may not be selected to be in a stripe to which data is being written. The system may also opt to not serve reads from a highly loaded node. For example, a "degraded mode read" may be performed, wherein a block in the highly loaded node is reconstructed from the other blocks of the same stripe. A degraded mode read is a read that is performed via the rest of the nodes in the same stripe, and the data is reconstructed via the failure protection. A degraded mode read may be performed when the read latency is too high, as the initiator of the read may assume that that node is down. If the load is high enough to create higher

read latencies, the cluster may revert to reading that data from the other nodes and reconstructing the needed data using the degraded mode read.

[0039] Each bucket manages its own distributed erasure coding instance (i.e., DFRAS 518) and does not need to cooperate with other buckets to perform read or write operations. There are potentially thousands of concurrent, distributed erasure coding instances working concurrently, each for the different bucket. This is an integral part of scaling performance, as it effectively allows any large filesystem to be divided into independent pieces that do not need to be coordinated, thus providing high performance regardless of the scale.

[0040] Each bucket handles all the file systems operations that fall into its shard. For example, the directory structure, file attributes and file data ranges will fall into a particular bucket's jurisdiction.

[0041] An operation done from any frontend starts by finding out what bucket owns that operation. Then the backend leader, and the node, for that bucket is determined. This determination may be performed by trying the last-known leader. If the last-known leader is not the current leader, that node may know which node is the current leader. If the last-known leader is not part of the bucket's penta-group anymore, that backend will let the front end know that it should go back to the configuration to find a member of the bucket's penta-group. The distribution of operations allows complex operations to be handled by a plurality of servers, rather than by a single computer in a standard system.

[0042] Each 4k block in the file is handled by an object called an extent. The extent ID is based on the inode ID and an offset (e.g., 1MB) within the file, and managed by the bucket. The extents are stored in the registry of each bucket. The extent ranges are protected by read-write locks. Each extent may manage the content of a contiguous

1MB for a file, by managing the pointers of up to 256 4k blocks that are responsible for that 1MB data.

[0043] **Write Operation**

[0044] At block 401, an extent in a VFS backend ("BE") receives a write request. The VFS frontend ("FE") calculates the bucket that is responsible for the extent for the 4k write (or aggregated writes). The FE may then send the blocks to the relevant bucket over the network, and that bucket will start handling the write. The block IDs that will be used to write these blocks are requested from the DFRAS in the Fig. 3. If the write contains enough 4k blocks to span several contiguous extents, the first extent receives the entire write request. At block 403, each extent that is part of the write locks the associated 4k blocks in order, for example from the first block of the first extent to the last block of the last extent. Locking is done on a complete 4k block basis, so even if the write specified new data for a portion of a 4k block the complete 4k region is blocked for coherency purposes. Write locking may be very "expensive" and is part of the reason why other systems have limited amount of writes they can perform when they only have a few metadata servers. Because the current disclosure allows an unlimited number of BE's, an unlimited number of 4k writes may be performed concurrently.

[0045] If the requested data does not fill a complete 4k block, each 4k block that is touched but not completely overridden is first read. The new data is overlaid over that block and complete 4k blocks may then be written to the DFRAS. For example, the user may write 5k data at an offset of 3584. The first 512 bytes are written on the first 4k block (as the last 512 bytes of it), the second 4k block is completely overridden, and the third 4k block is overridden for its first 512 bytes. The first and third 4k blocks will be first read, the new data will override the correct portions of it, and then three full 4k blocks will actually be written to the DFRAS.

[0046] Each block has its EEDP (end-to-end data protection) calculated before it's being written to the DFRAS, and the EEDP information may also be given to the DFRAS so it can make sure that each written block reaches the final NVM destination unaltered with the correct EEDP. If a bit flip occurred over the network or a logical error happened, the end node will receive 4k block data that calculates to a different EEDP data, and thus the DFRAS will know that it needs to resend that data again.

[0047] At block 405, once an extent acknowledges the lock, writing may begin. The write lock allows only a single write at a time over each portion of the extent. Two concurrent writes may occur over different ranges of the extent. If an extents is on another bucket, a remote procedure (RPC) call may be used to cause the write to execute in an address space as if it were a local write, e.g., without explicitly coding the details for the remote interaction.

[0048] At block 407, the journal is saved to the DFRAS. If possible, the first block written may be compressed to fit the journal. The journal (which is either its own block or within the first block) is updated with the returned block IDs. The first few bytes of the written blocks may actually be replaced by the backpointer to the extent, which later (along with the EEDP) allows the system to make sure that the read data is intact.

[0049] At block 409, if the data to be written is not 4k aligned, or not 4k multiple in size, the data may be padded on either (or both) ends of the buffer to a 4k multiple. If the data is padded, existing data is read from the first and/or last block (same block if the write is less than 4k) and the requested written data is padded with the read data, such that complete 4k blocks can be written. In this way small files can be united in the inode and the extent.

[0050] At block 411, after the DFRAS are written, the extents are updated with the block IDs, the EEDP, and also the first few bytes of data for all the blocks. The extent may be stored in the registry, and will be destaged to the DFRAS on the next destaging of

the registry. At block 413, once all the writes are done, the extent ranges are unlocked in reverse order to the locking order. Once the registry is updated, that data may be read.

[0051] **Append Operation**

[0052] A special form of a write is an *append*. With append, systems gets a blob of data that may be appended at the end of the current data. Doing so from one host with a single writer is not difficult – just find the last offset and perform a write operation. However, append operations may happen concurrently from several FEs, and the system must make sure that each append is consistent within itself. For example, two large append operations must not be allowed to interleave data.

[0053] The inode knows which extent is the last extent for the file, and the extent also knows it is the last extent (only one extent may be last). Only the last extent will accept append operations. Any other extent will reject them.

[0054] On each append operations the FE goes through the inode to find which is the last extent. If it hit an extent that refused to perform the operation, the FE will go through the inode again to look for the last extent.

[0055] When an extent performs appends, it perform each append in full in the order they were received, so when each append start executing the extent finds the last block of data and performs the write (after locking that block). If the append ends after that last extent, the extent will create a new extent in a special mode that may execute just one append before it is "open for use." The extent may then forward the remaining pieces of the data to that extent. That extent becomes the last extent and is marked as open for use. The extent that is not last anymore, will fail all remaining append operations. The remaining append operations will therefore go back to the FE to find who is the last extent from the inode and continue normally. If the write is large enough

to span several extents, the last extent will create all extents required in that mode and will forward the correct data to be written to the created extents. Once the all the data is written to all extents, the original extent will mark the last extent of the append as the last and will mark all extents as open for use. The original extent will then go through all pending append operations, which will fail and be sent the originating FE to restart the operation and receive the correct extent.

[0056] The semantics of the operations are not affected by the order of concurrent appends. However, it must be ensured that no data will be interleaved between the different writes. If the FE or the inode would find the last offset and perform a standard write, there will be races and the file will end up with inconsistent data in relation to the append operation semantics.

[0057] **Read Operation**

[0058] At block 501, the extent in the VFS frontend receives the read request. It calculates and finds out what bucket is responsible for the extent that contains the read, then finds out what backend (node) currently runs that bucket. Then it sends the read request over RPC to that bucket. If a read operation spans more than one extent (more than one continuous 1MB), the VFS FE may split it into several distinct read operations that will then be aggregated back by the VFS FE. Each such read operation may request the read to happen from a single extent. The VFS FE then sends a read request to the node responsible to the bucket holding that extent.

[0059] At block 503, the VFS BE holding the bucket looks up an extent key in the registry. If the extent does not exist, a buffer of zeros is returned at block 505. At block 507, if the extent does exist, the 4k blocks in the extent are read locked and data is read from the DFRAS. At block 509, the registry will overlay a corresponding shadow from of the data from VM (if one exists) over the stored data and return the actual content

pointed to by the extent. The shadow of the extent comprises data changes that have not yet been committed to non-volatile memory (e.g., flash SSD). If the requested data is not 4k aligned or not 4k in size, the data is retrieved from the persistent distributed coding scheme, and then the ends may be "chopped" to meet the requested (offset, size).

[0060] At block 511, each received block is checked for the backpointer (pointing to the block ID of the extent) and the EEDP (end-to-end-data-protection). If either check fails, the extent requests the blocks from the distributed erasure coding of the bucket to do a degraded mode read at block 513 in which the other blocks of the stripe are read in order to reconstruct the requested block. If the extent exists, it has a list of all the blocks of a stripe that a persistent layer stored. The reconstructed block is checked for the backpointer (pointing to the block ID of the extent) and the EEDP (end-to-end-data-protection) at block 515. At block 517, if either check fails, an IO Error is returned. Once the correct data is confirmed at block 511, it is returned to the requesting FE, after the first few bytes are replaced with the original block data that was stored in the extent and override the backpointer; a new write to the distributed erasure coding system (DECS) is initiated; and the extent with the new block ID also is updated.

[0061] FIG. 6 illustrates an example implementation in which two distributed failure resilient address spaces reside on a plurality of solid-state storage disks. The chunks $510_{1,1}$–$510_{D,C}$ may be organized into a plurality of chunk stripes 520_1–520_S (S being an integer). In an example implementation, each chunk stripe 520_s (s being an integer, where $1 \leq s \leq S$) is separately protected using forward error correction (e.g., erasure coding). The number of chunks $510_{d,c}$ in any particular chunk stripe 520_s may thus be determined based on the desired level of data protection.

[0062] Assuming, for purposes of illustration, that each chunk stripe 520_s comprises N = M + K (where each of N, M, and K are integers) chunks $510_{d,c}$, then M of the N chunks $510_{d,c}$ may store data digits (typically binary digits or "bits" for current storage devices)

and K of the N chunks $510_{d,c}$ may store protection digits (again, typically bits). To each stripe 520_s, then, the virtual file system may assign N chunks $510_{d,c}$ from N different failure domains.

[0063] As used herein, a "failure domain" refers to a group of components in which a failure of any single one of the components (the component losing power, becoming nonresponsive, and/or the like) may result in failure of all the components. For example, if a rack has a single top-of-the-rack switch a failure of that switch will bring down connectivity to all the components (e.g., compute, storage, and/or VFS nodes) on that rack. Thus, to the rest of the system it is equivalent to if all of the components on that rack failed together. A virtual file system in accordance with this disclosure may comprise fewer failure domains than chunks 510.

[0064] In an example implementation where the nodes of the virtual file system are connected and powered in a fully-redundant way with only a single storage device 506 per such node, a failure domain may be just that single storage device 506. Thus, in an example implementation, each chunk stripe 520_s comprises a plurality of chunks $510_{d,c}$ residing on each of N of the storage devices 506_1–506_D., (D is thus greater than or equal to N). An example of such an implementation is shown in FIG 7.

[0065] In FIG. 6, D = 7, N = 5, M = 4, K = 1, and the storage is organized into two DFRASs. These numbers are merely for illustration and not intended as limiting. Three chunk stripes 520 of the first DFRAS are arbitrarily called out for illustration. The first chunk stripe 520_1 consists of chunks $510_{1,1}$, $510_{2,2}$, $510_{3,3}$, $510_{4,5}$ and $510_{5,6}$; the second chunk stripe 520_2 consists of chunks $510_{3,2}$, $510_{4,3}$, $510_{5,3}$, $510_{6,2}$ and $510_{7,3}$; and the third chunk stripe 520_3 consists of chunks $510_{1,4}$, $510_{2,4}$, $510_{3,5}$, $510_{5,7}$ and $510_{7,5}$.

[0066] Although D = 7 and N = 5 in the simple example of FIG. 6 in an actual implementation D may be much larger than N (e.g., by a multiple of an integer greater than 1 and possibly as high as many orders of magnitude) and the two values may be

chosen such that the probability of any two chunk stripes 520 of a single DFRAS residing on the same set of N storage devices 506 (or, more generally, on the same set of N failure domains) is below a desired threshold. In this manner, failure of any single storage device 506_d (or, more generally, any single failure domain) will result (with the desired statistical probably determined based on: chosen values of D and N, the sizes of the N storage devices 506, and the arrangement of failure domains) in loss of at most one chunk $510_{b,c}$ of any particular stripe 520_s. Even further, a dual failure will result in vast majority of stripes losing at most a single chunk $510_{b,c}$ and only small number of stripes (determined based on the values of D and N) will lose two chunks out of any particular stripe 520_s (e.g., the number of two-failure stripes may be exponentially less than the number of one-failure stripes).

[0067] For example, if each storage device 506_d is 1TB, and each chunk is 128MB, then failure of storage device 506_d will result (with the desired statistical probably determined based on: chosen values of D and N, the sizes of the N storage devices 506, and the arrangement of failure domains) in 7812 (= 1TB/128MB) chunk stripes 520 losing one chunk 510. For each such affected chunk stripe 520_s, the lost chunk $510_{d,c}$ can be quickly reconstructed using an appropriate forward error correction algorithm and the other N-1 chunks of the chunk stripe 520_s. Furthermore, since the affected 7812 chunk stripes are uniformly distributed across all of the storage devices 506_1–506_D, reconstructing the lost 7812 blocks $510_{d,c}$ will involve (with the desired statistical probably determined based on: chosen values of D and N, the sizes of the N storage devices 506, and the arrangement of failure domains) reading the same amount of data from each of storage devices 506_1–506_D (i.e., the burden of reconstructing the lost data is uniformly distributed across all of storage devices 506_1–506_D so as to provide for very quick recovery from the failure).

[0068] Next, turning to the case of a concurrent failure of two of the storage devices 506_1–506_D (or, more generally, concurrent failure of two failure domains), due to the uniform distribution of the chunk stripes 520_1–520_S of each DFRAS over all of the storage devices 506_1–506_D, only a very small number of chunk stripes 520_1–520_S will have lost two of their N chunks. The virtual file system may be operable to quickly identify such two-loss chunk stripes based on metadata which indicates a mapping between chunk stripes 520_1–520_S and the storage devices 506_1–506_D. Once such two-loss chunk stripes are identified, the virtual file system may prioritize reconstructing those two-loss chunk stripes before beginning reconstruction of the one-loss chunk stripes. The remaining chunk stripes will have only a single lost chunk and for them (the vast majority of the affected chunk stripes) a concurrent failure of two storage devices 506_d is the same as a failure of only one storage device 506_d. Similar principles apply for a third concurrent failure (the number of chunk stripes having three failed blocks will be even less than the number having two failed blocks in the two concurrent failure scenario), and so on. In an example implementation, the rate at which reconstruction of a chunk stripe 520_s is performed may be controlled based on the number of losses in the chunk stripe 520_s. This may be achieved by, for example, controlling the rates at which reads and commits for reconstruction are performed, the rates at which FEC computations for reconstruction are performed, the rates at which network messages for reconstruction are communicated, etc.

[0069] FIG. 7 illustrates a forward error correction scheme which may be used for protecting data stored to nonvolatile memory of a virtual file system in accordance with an example implementation of this disclosure. Shown are storage blocks $902_{1,1}$–$902_{7,7}$ of block stripes 530_1–530_4 of a DFRAS. In the protection scheme of FIG. 7, five blocks of each stripe are for storage of data digits and two blocks of each stripe are for data storage of protection digits (i.e., M=5 and K=2). In FIG. 7, the protection digits are calculated using the following equations (1)–(9):

$$P1 = D1_1 \oplus D2_2 \oplus D3_3 \oplus D4_4 \oplus D5_4 \quad (1)$$

$$P2 = D2_1 \oplus D3_2 \oplus D4_3 \oplus D5_3 \oplus D1_4 \quad (2)$$

$$P3 = D3_1 \oplus D4_2 \oplus D5_2 \oplus D1_3 \oplus D2_4 \quad (3)$$

$$P4 = D4_1 \oplus D5_1 \oplus D1_2 \oplus D2_3 \oplus D3_4 \quad (4)$$

$$Z = D5_1 \oplus D5_2 \oplus D5_3 \oplus D5_4 \quad (5)$$

$$Q1 = D1_1 \oplus D1_2 \oplus D1_3 \oplus D1_4 \oplus Z \quad (6)$$

$$Q2 = D2_1 \oplus D2_2 \oplus D2_3 \oplus D2_4 \oplus Z \quad (7)$$

$$Q3 = D3_1 \oplus D3_2 \oplus D3_3 \oplus D3_4 \oplus Z \quad (8)$$

$$Q4 = D4_1 \oplus D4_2 \oplus D4_3 \oplus D4_4 \oplus Z \quad (9)$$

[0070] Thus, the four stripes 530_1–530_4 in FIG. 7 are part of a multi-stripe (four stripes, in this case) FEC protection domain and loss of any two or fewer blocks in any of the block stripes 530_1–530_4 can be recovered from using various combinations of the above equations (1)–(9). For comparison, an example of a single-stripe protection domain would be if $D1_1$, $D2_2$, $D3_3$, $D4_4$, $D5_4$ were protected only by P1 and $D1_1$, $D2_2$, $D3_3$, $D4_4$, $D5_4$, and P1 were all written to stripe 530_1 (530_1 would be the single-stripe FEC protection domain).

[0071] In accordance with an example implementation of this disclosure, a plurality of computing devices are communicatively coupled to each other via a network, and each of the plurality of computing devices comprises one or more of a plurality of storage devices. A plurality of failure resilient address spaces are distributed across the plurality of storage devices such that each of the plurality of failure resilient address spaces spans a plurality of the storage devices. Each one of the plurality of failure resilient address spaces is organized into a plurality of stripes (e.g., a plurality of 530 as in FIGs. 6 and 7). Each one or more stripes of the plurality of stripes is part of a

respective one of a plurality of forward error correction (FEC) protection domains (e.g., a multi-stripe FEC domain such as in FIG. 7). Each of the plurality of stripes may comprise a plurality of storage blocks (e.g., a plurality of 512). Each block of a particular one of the plurality of stripes may reside on a different one of the plurality of storage devices. A first portion the plurality of storage blocks (e.g., the quantity of five consisting of $902_{1,2}$–$902_{1,6}$ of stripe 530_1 of FIG. 7) may be for storage of data digits, and a second portion of the plurality of storage blocks (e.g., the quantity of two $902_{1,1}$ and $902_{1,7}$ of stripe 530_1 of FIG. 7) may be for storage of protection digits calculated based, at least in part, on the data digits. The plurality of computing devices may be operable to rank the plurality of stripes. The rank may be used for selection of which of the plurality of stripes to use for a next commit operation to the one of the plurality of failure resilient address spaces. The rank may be based on how many protected and/or unprotected storage blocks are in each of the plurality of stripes. For any particular one of the plurality of stripes, the rank may be based on a bitmap stored on the plurality of storage devices with the particular one of the plurality of stripes. The rank may be based on how many blocks currently storing data are in each of the plurality of stripes. The rank may be based on read and write overhead for committing to each of the plurality of stripes. Each of the failure resilient address spaces may be owned by only one of the plurality of computing devices at any given time, and each one of the plurality of failure resilient address spaces may be read and written only by its owner. Each of the computing devices may own multiple of the failure resilient address spaces. The plurality of storage devices may be organized into a plurality of failure domains. Each one of the plurality of stripes may span a plurality of the failure domains. Each of the failure resilient address spaces may span all of the plurality of failure domains, such that upon failure of any particular one of the plurality of failure domains, a workload for reconstructing the lost data is distributed among each of the others of the plurality of failure domains. The plurality of stripes may be distributed across the plurality of failure domains such that, in

the event of concurrent failure of two of the plurality of failure domains, the chances of two blocks of any particular one of the plurality stripes residing on the failed two of the plurality of failure domains is exponentially less than the chances of only one block of any particular one of the plurality stripes residing on the failed two of the plurality of failure domains. The plurality of computing devices may be operable to first reconstruct any of the plurality of stripes which have two failed blocks, and then reconstruct any of the plurality of stripes which have only one failed block. The plurality of computing devices may be operable to perform the reconstruction of the plurality of stripes which have two failed blocks at a higher rate (e.g., with a greater percentage of CPU clock cycles dedicated to the reconstruction, a greater percentage of network transmit opportunities dedicated to the reconstruction, and/or the like.) than the rate of reconstruction of the plurality of stripes which have only one failed block. The plurality of computing devices may be operable to determine, in the event of a failure of one or more of the failure domains, a rate at which any particular lost block is reconstructed based on how many other blocks of a same one of the plurality of stripes have been lost. Wherein one or more of the plurality of failure domains comprises a plurality of the storage devices. Each of the plurality of FEC protection domains may span multiple stripes of the plurality of stripes. The plurality of stripes may be organized into a plurality of groups (e.g., chunk stripes 520_1–520_S as in FIG. 6), where each of the plurality of groups comprises one or more of the plurality of stripes, and, the plurality of computing devices are operable to rank, for each of the groups, the one or more of the plurality of stripes of the group. The plurality of computing devices may be operable to: perform successive committing operations to a selected one of the plurality of groups until the one or more of the plurality of stripes of the of the group no longer meets a determined criterion, and upon the selected one of the plurality of groups no longer meeting the determined criterion, select a different one of the plurality of groups. The criterion may be based on how many blocks are available for new data to be written to.

[0072] While the present method and/or system has been described with reference to certain implementations, it will be understood by those skilled in the art that various changes may be made and equivalents may be substituted without departing from the scope of the present method and/or system. In addition, many modifications may be made to adapt a particular situation or material to the teachings of the present disclosure without departing from its scope. Therefore, it is intended that the present method and/or system not be limited to the particular implementations disclosed, but that the present method and/or system will include all implementations falling within the scope of the appended claims.

[0073] As utilized herein the terms "circuits" and "circuitry" refer to physical electronic components (i.e. hardware) and any software and/or firmware ("code") which may configure the hardware, be executed by the hardware, and or otherwise be associated with the hardware. As used herein, for example, a particular processor and memory may comprise first "circuitry" when executing a first one or more lines of code and may comprise second "circuitry" when executing a second one or more lines of code. As utilized herein, "and/or" means any one or more of the items in the list joined by "and/or". As an example, "x and/or y" means any element of the three-element set {(x), (y), (x, y)}. In other words, "x and/or y" means "one or both of x and y". As another example, "x, y, and/or z" means any element of the seven-element set {(x), (y), (z), (x, y), (x, z), (y, z), (x, y, z)}. In other words, "x, y and/or z" means "one or more of x, y and z". As utilized herein, the term "exemplary" means serving as a non-limiting example, instance, or illustration. As utilized herein, the terms "e.g.," and "for example" set off lists of one or more non-limiting examples, instances, or illustrations. As utilized herein, circuitry is "operable" to perform a function whenever the circuitry comprises the necessary hardware and code (if any is necessary) to perform the function, regardless of whether performance of the function is disabled or not enabled (e.g., by a user-configurable setting, factory trim, etc.).

CLAIMS

What is claimed is:

1. A system comprising:

a plurality of storage devices, wherein a failure resilient address space is distributed across the plurality of storage devices such that the failure resilient address space spans more than one storage device of the plurality of storage devices; and

a plurality of computing devices communicatively coupled to each other and to the plurality of storage devices via a network, wherein each of the plurality of computing devices is operable to perform a file operation over a plurality of memory blocks, and wherein each of the plurality of computing devices is operable to maintain metadata that maps the plurality of memory blocks to the failure resilient address space.

2. The system of claim 1, wherein the plurality of storage devices comprises non-volatile memory.

3. The system of claim 1, wherein the file operation comprises writing data into the plurality of memory blocks.

4. The system of claim 3, wherein a buffer of data to be written to a first memory block of the plurality of memory blocks is compressed to provide space for a journal associated with the data written into the plurality of memory blocks.

5. The system of claim 3, wherein a buffer of data to be written to a memory block of the plurality of memory blocks is padded with data previously written into a portion of the memory block.

6. The system of claim 3, wherein in coordination with writing data to a memory block of the plurality of memory blocks, an extent is updated, and wherein the updated extent comprises an identification of a plurality of blocks associated with protecting the data being written to the memory block.

7. The system of claim 1, wherein the file operation comprises reading data from the plurality of memory blocks.

8. The system of claim 7, wherein a data read from a particular memory block of the plurality of memory blocks is checked for errors using a distributed erasure code based on blocks identified in an extent.

9. The system of claim 8, wherein a degraded data read is performed when the particular memory block of the plurality of memory blocks is found to be in error.

10. The system of claim 9, wherein the degraded data read comprises regenerating the particular memory block from one or more memory blocks other than the particular memory block of the plurality of memory blocks.

11. A method for accessing storage media, the method comprising:

distributing a failure resilient address space across a plurality of storage devices, wherein the distribution is performed by a plurality of computing devices;

operating on a file in a plurality of memory blocks in the plurality of storage devices, wherein the operation is performed by the plurality of computing devices; and

maintaining metadata within the plurality of computing devices to map a plurality of memory blocks to the failure resilient address space.

12. The method of claim 11, wherein the plurality of storage devices comprises non-volatile memory.

13. The method of claim 11, wherein operating on the file comprises writing data into the plurality of memory blocks.

14. The method of claim 13, wherein writing data comprises compressing a buffer of data to be written to a first memory block of the plurality of memory blocks to provide space for a journal associated with the data written into the plurality of memory blocks.

15. The method of claim 13, wherein writing data comprises padding a buffer of data to be written to a memory block of the plurality of memory blocks with data previously written into a portion of the memory block.

16. The method of claim 13, wherein in coordination with writing data to a memory block of the plurality of memory blocks, the method comprises updating an extent such that the extent comprises an identification of a plurality of blocks associated with protecting the data being written to the memory block.

17. The method of claim 11, wherein operating on the file comprises reading data from the plurality of memory blocks.

18. The method of claim 17, wherein the method comprises checking data read from a particular memory block of the plurality of memory blocks for errors using a distributed erasure code based on blocks identified in an extent.

19. The method of claim 18, wherein the method comprises performing a degraded data read when the particular memory block of the plurality of memory blocks is found to be in error.

20. The method of claim 19, wherein performing a degraded data read comprises regenerating the particular memory block from one or more memory blocks other than the particular memory block of the plurality of memory blocks.

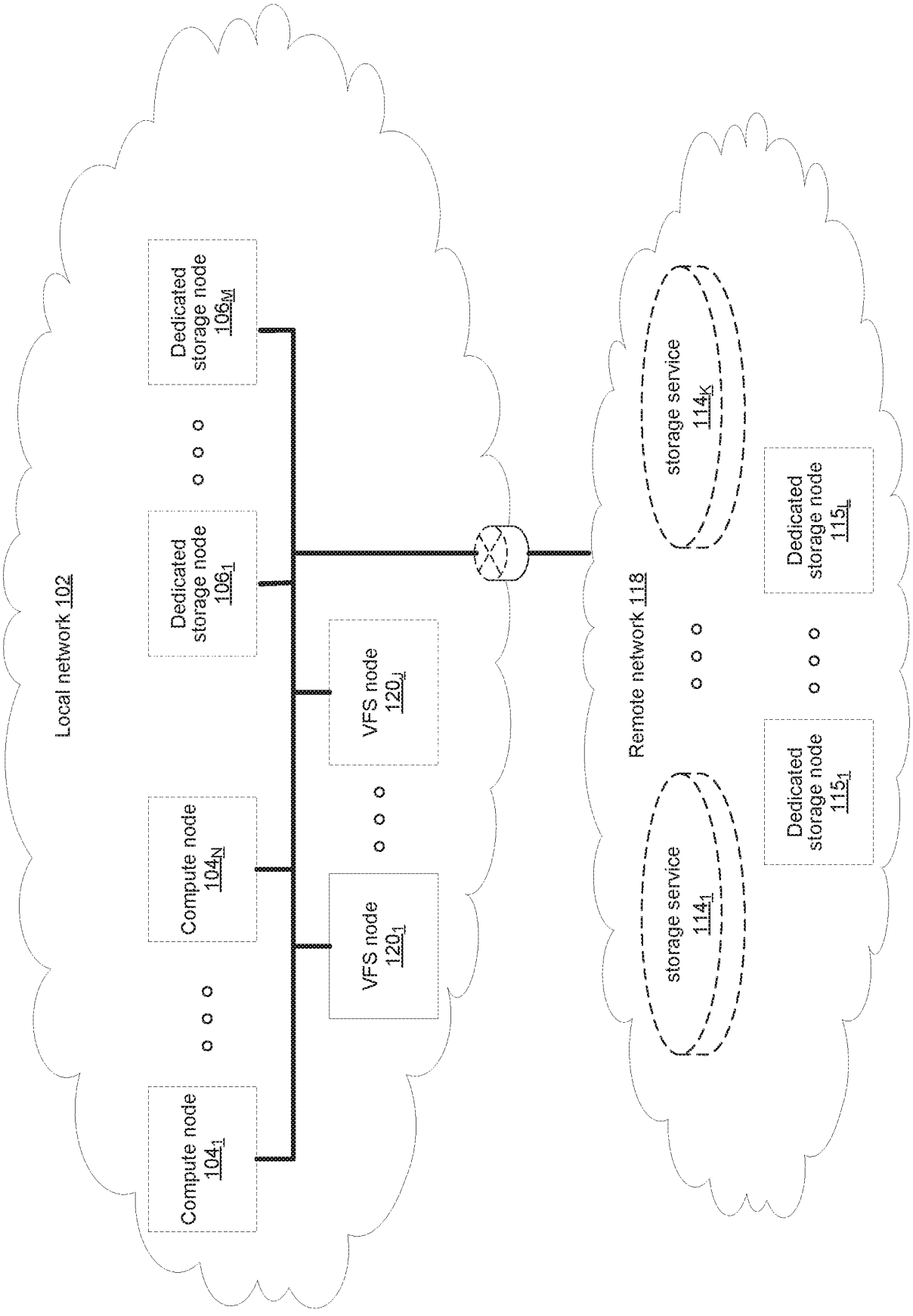
Local network 102
Compute node 104_1
Compute node 104_N
Dedicated storage node 106_1
Dedicated storage node 106_M
VFS node 120_1
VFS node 120_J
Remote network 118
storage service 114_1
storage service 114_K
Dedicated storage node 115_1
Dedicated storage node 115_L

FIG. 1

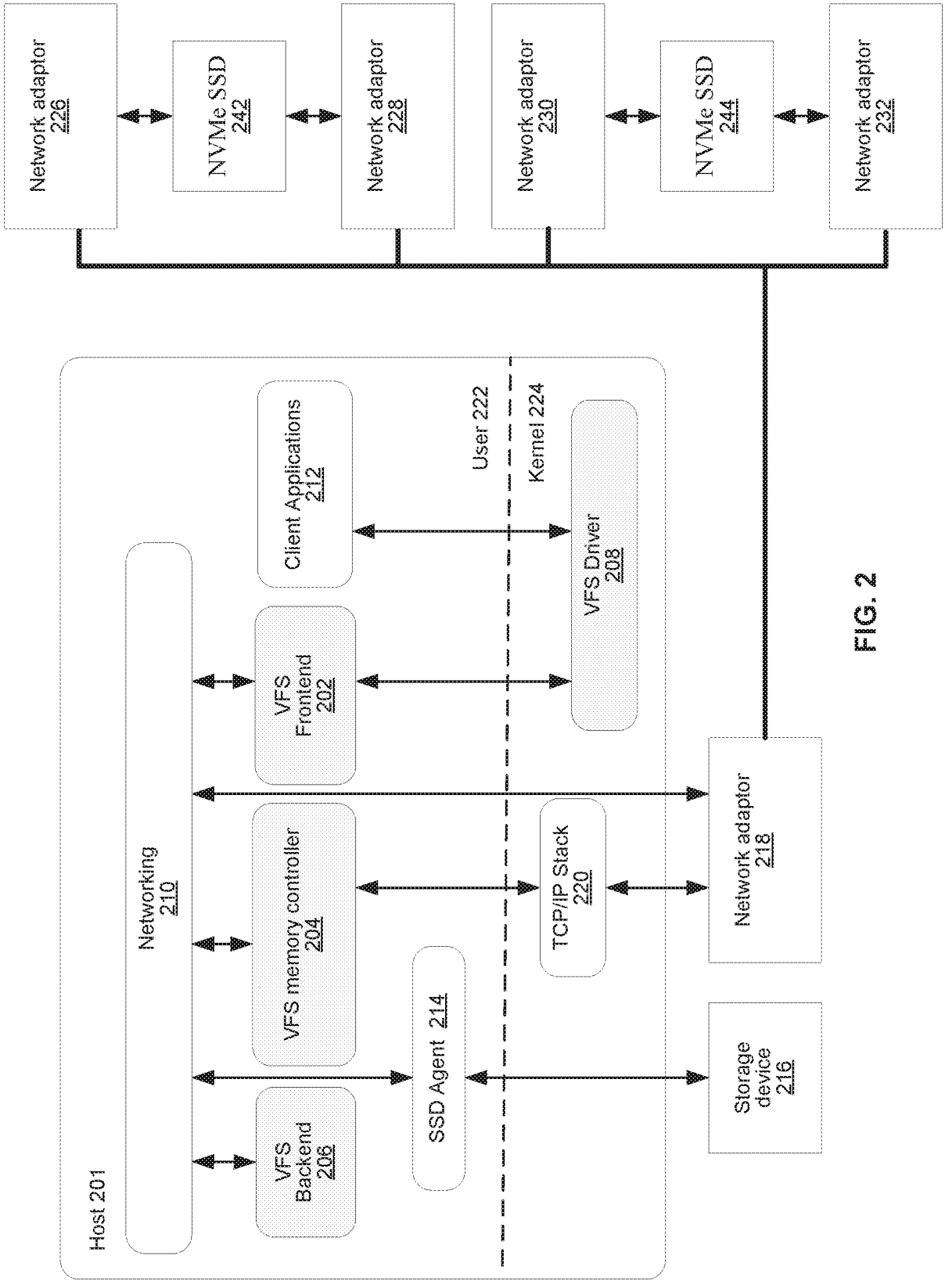
Network adaptor 226
NVMe SSD 242
Network adaptor 228
Network adaptor 230
NVMe SSD 244
Network adaptor 232
Host 201
Networking 210
Client Applications 212
User 222
Kernel 224
VFS Driver 208
VFS Frontend 202
VFS memory controller 204
TCP/IP Stack 220
Network adaptor 218
VFS Backend 206
SSD Agent 214
Storage device 216

FIG. 2

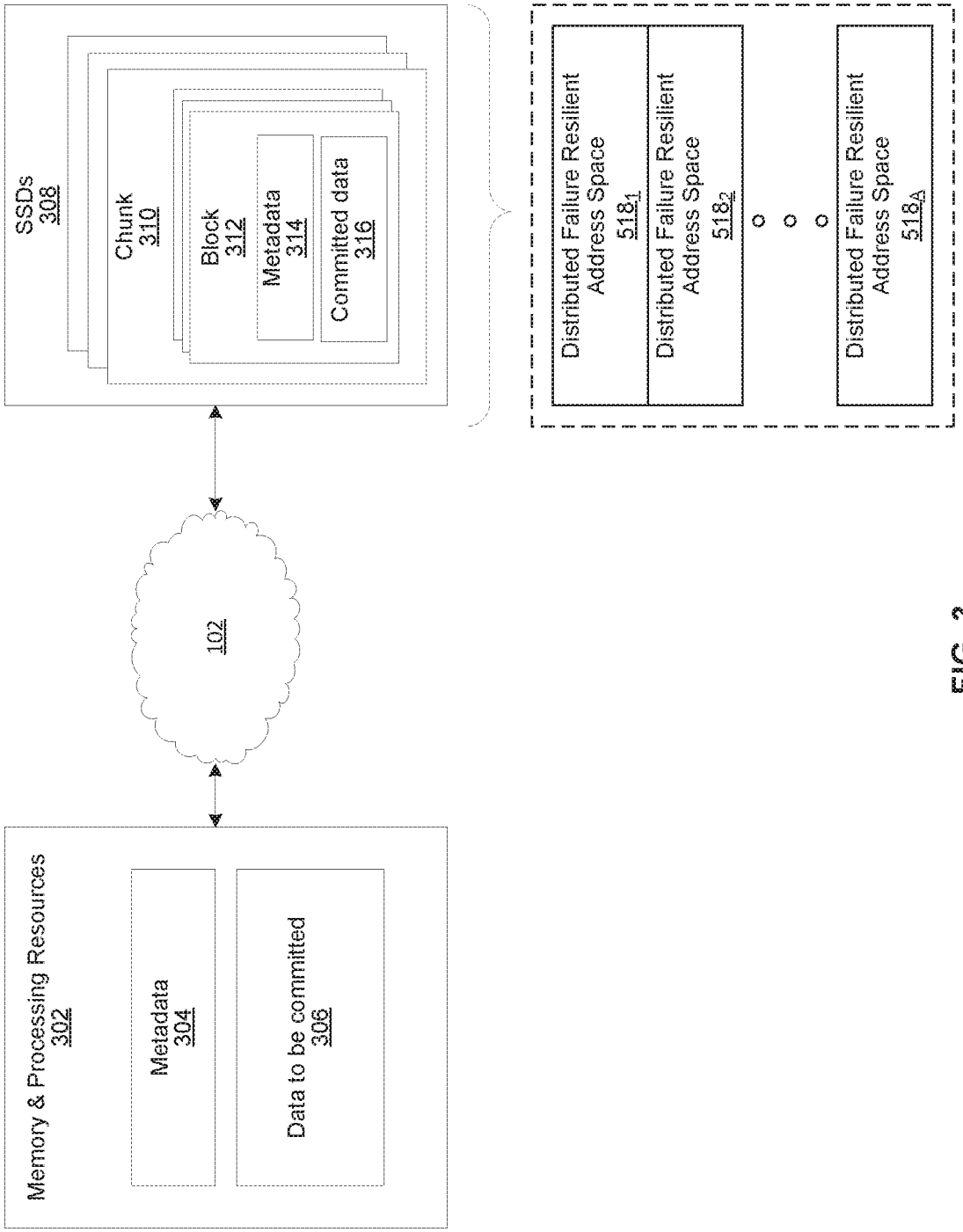
SSDs
308
Chunk
310
Block
312
Metadata
314
Committed data
316
Distributed Failure Resilient Address Space
518_1
Distributed Failure Resilient Address Space
518_2
Distributed Failure Resilient Address Space
518_A
102
Memory & Processing Resources
302
Metadata
304
Data to be committed
306

FIG. 3

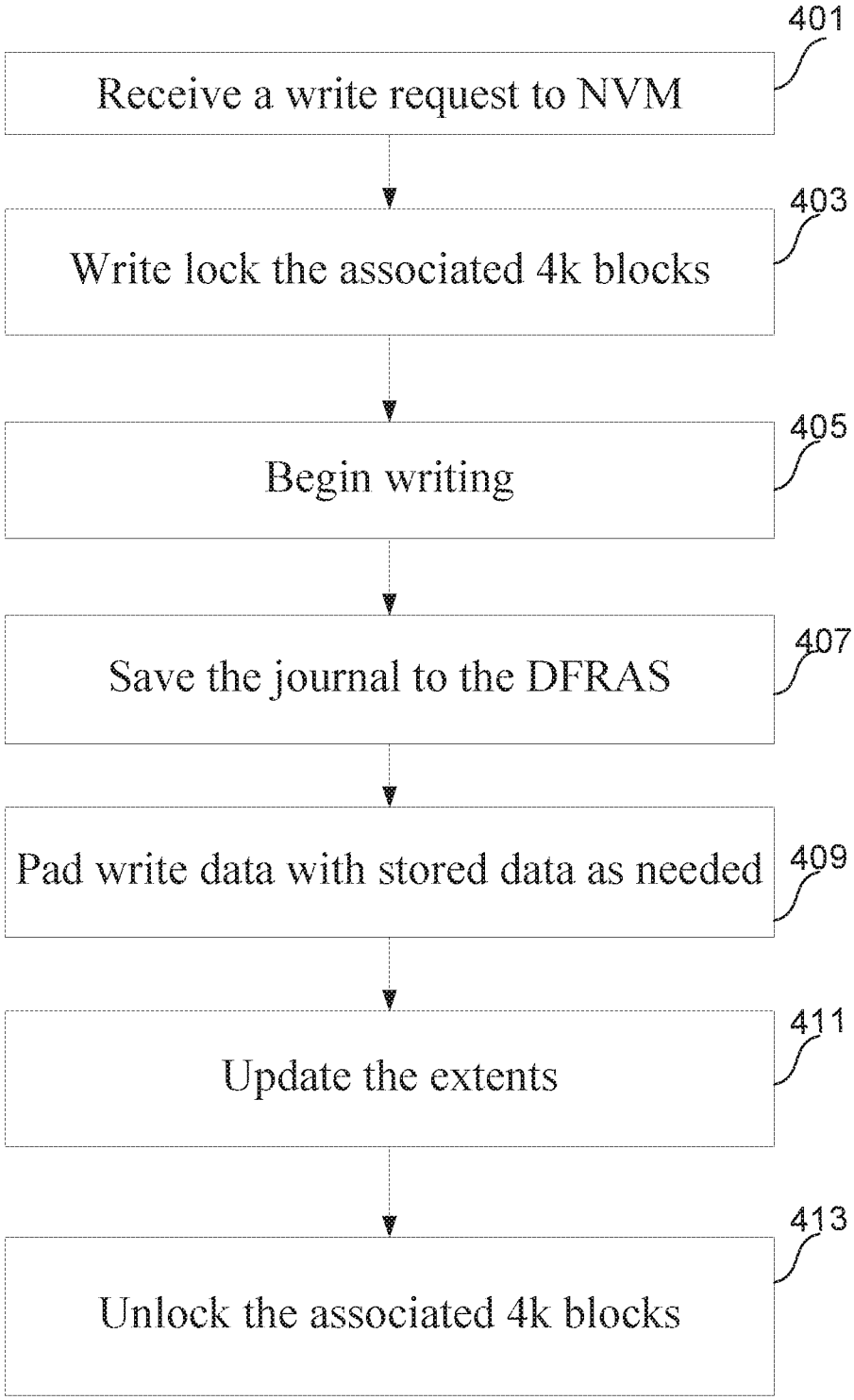
401
Receive a write request to NVM
403
Write lock the associated 4k blocks
405
Begin writing
407
Save the journal to the DFRAS
409
Pad write data with stored data as needed
411
Update the extents
413
Unlock the associated 4k blocks

FIG. 4

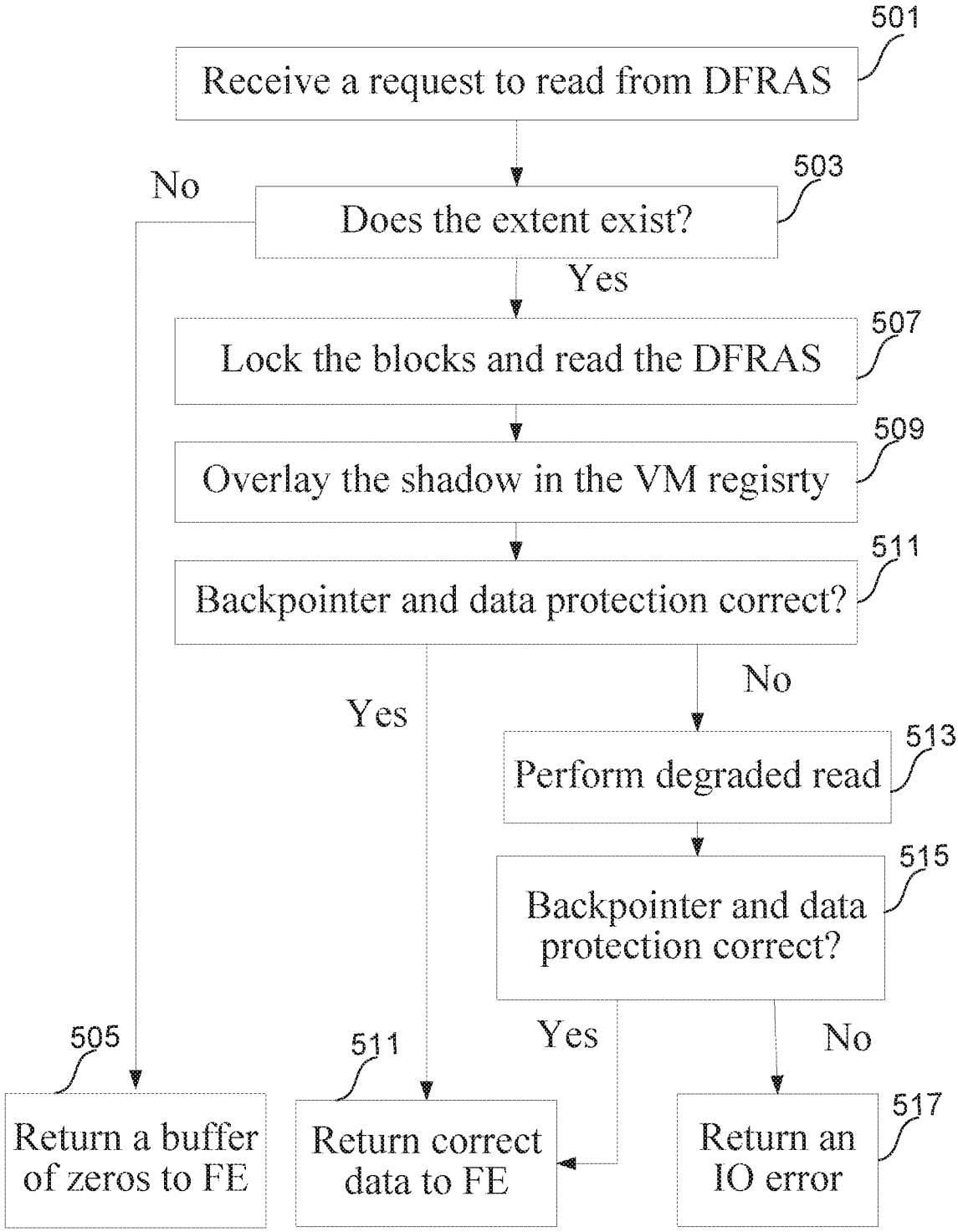
501
Receive a request to read from DFRAS
503
Does the extent exist?
No
Yes
507
Lock the blocks and read the DFRAS
509
Overlay the shadow in the VM regisrty
511
Backpointer and data protection correct?
No
Yes
513
Perform degraded read
515
Backpointer and data protection correct?
Yes
No
505
Return a buffer of zeros to FE
511
Return correct data to FE
517
Return an IO error

FIG. 5

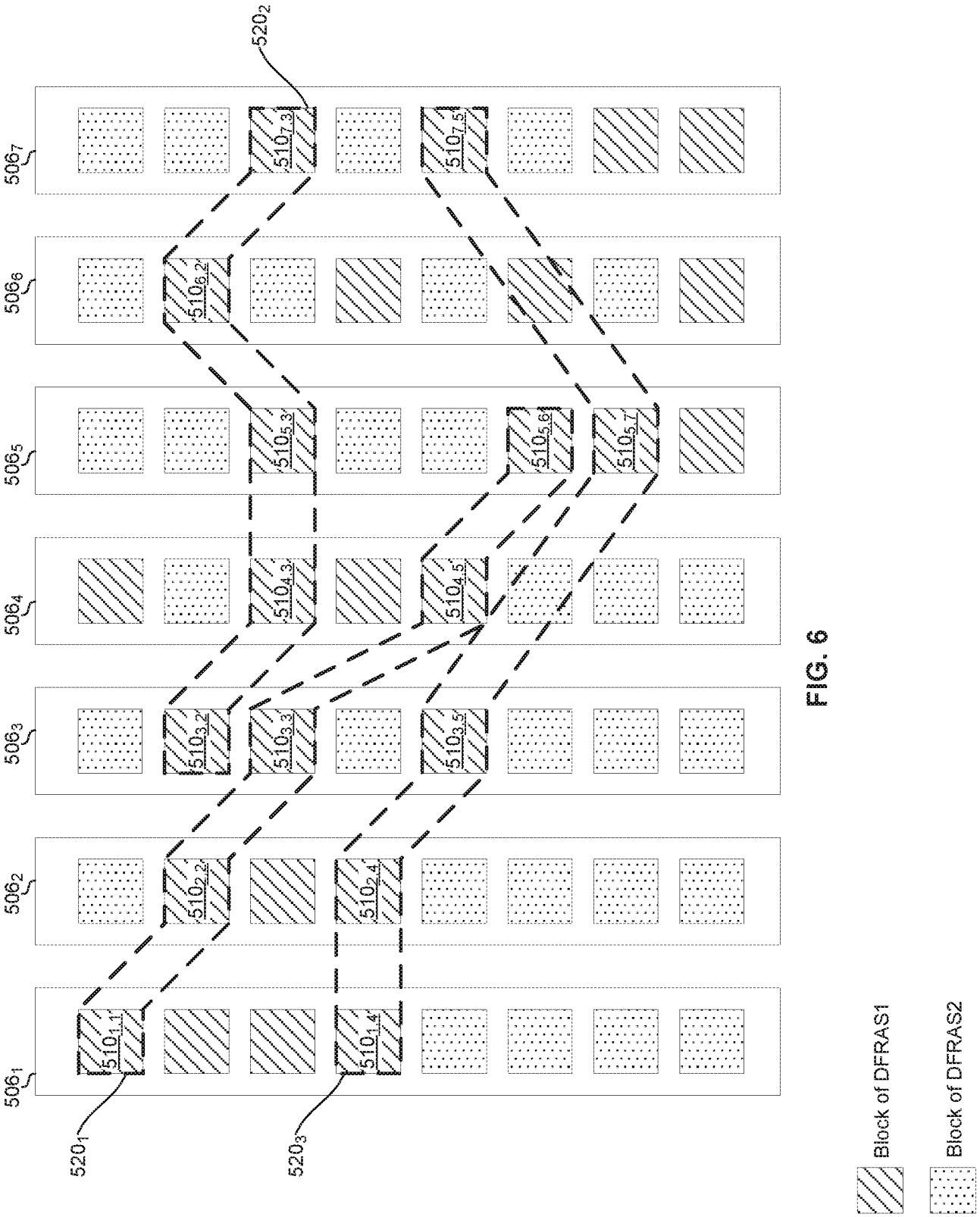
506_1
506_2
506_3
506_4
506_5
506_6
506_7
520_1
520_2
520_3
$510_{1,1}$
$510_{1,4}$
$510_{2,2}$
$510_{2,4}$
$510_{3,2}$
$510_{3,3}$
$510_{3,5}$
$510_{4,3}$
$510_{4,5}$
$510_{5,3}$
$510_{5,6}$
$510_{5,7}$
$510_{6,2}$
$510_{7,3}$
$510_{7,5}$
Block of DFRAS1
Block of DFRAS2

FIG. 6

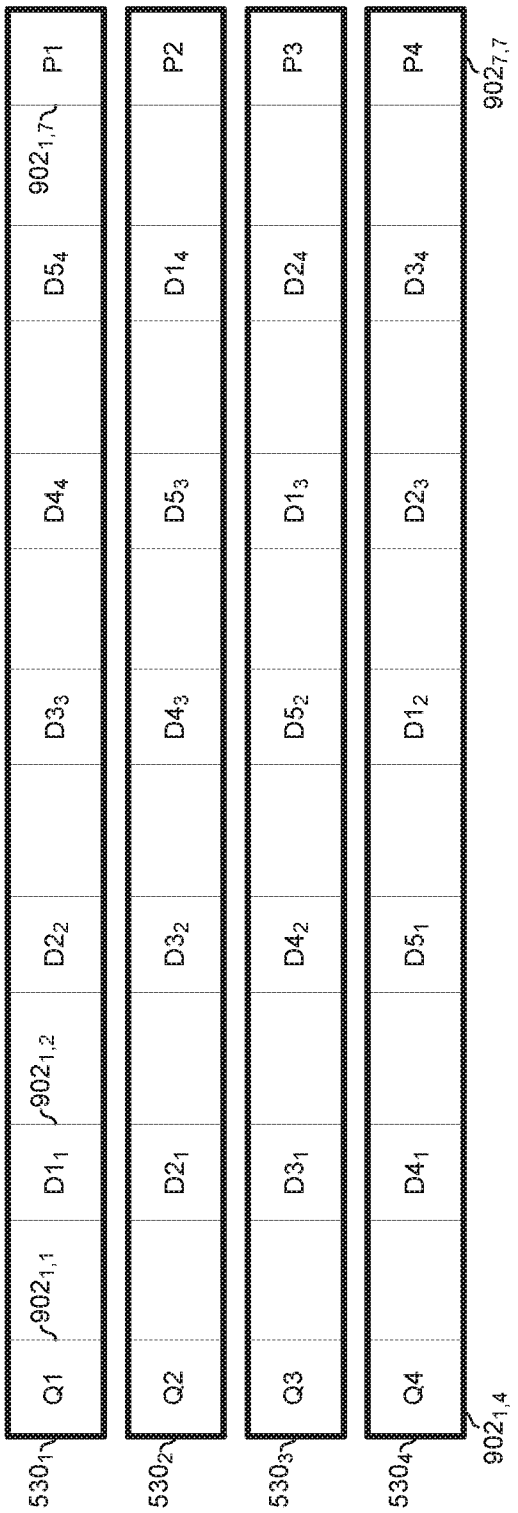
530_1
530_2
530_3
530_4
Q1
Q2
Q3
Q4
$902_{1,1}$
$902_{1,2}$
$902_{1,4}$
$902_{1,7}$
$902_{7,7}$
$D1_1$
$D2_1$
$D3_1$
$D4_1$
$D2_2$
$D3_2$
$D4_2$
$D5_1$
$D3_3$
$D4_3$
$D5_2$
$D1_2$
$D4_4$
$D5_3$
$D1_3$
$D2_3$
$D5_4$
$D1_4$
$D2_4$
$D3_4$
P1
P2
P3
P4

FIG. 7

INTERNATIONAL SEARCH REPORT

International application No.
PCT/IB18/01316

A. CLASSIFICATION OF SUBJECT MATTER
IPC - G06F 12/02, 12/16, 11/07 (2019.01)
CPC - G06F 12/02, 12/16, 11/073, 11/0727

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
See Search History document

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
See Search History document

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
See Search History document

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X --- Y	US 2016/0041878 A1 (PURE STORAGE, INC.) 11 February 2016; abstract; figure 3; paragraphs [0018], [0019], [0026], [0033], [0034], [0046]	1-3, 7-13, 17-20 --- 4-6, 14-16
Y	US 2005/0144387 A1 (ADL-TABATABAI, A et al.) 30 June 2005; abstract; paragraphs [0064], [0069]	4, 14
Y	US 6034831 A (DOBBEK, J et al.) 07 March 2000; claims 1, 7	5, 15
Y	US 2017/0206221 A1 (COMMVAULT SYSTEMS, INC.) 20 July 2017; paragraphs [0005], [0274], [0277]	6, 16

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

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
15 February 2019 (15.02.2019)	27 FEB 2019
Name and mailing address of the ISA/ Mail Stop PCT, Attn: ISA/US, Commissioner for Patents P.O. Box 1450, Alexandria, Virginia 22313-1450 Facsimile No. 571-273-8300	Authorized officer Shane Thomas PCT Helpdesk: 571-272-4300 PCT OSP: 571-272-7774

Form PCT/ISA/210 (second sheet) (January 2015)