The Journal of Supercomputing (2020) 76:174–203
https://doi.org/10.1007/s11227-019-03017-y

Elastic HDFS: interconnected distributed architecture for availability–scalability enhancement of large-scale cloud storages

M. Maghsoudloo[1] · **N. Khoshavi**[2]

Published online: 14 October 2019

Abstract
This paper presents an interconnected distributed architecture for storing data and metadata in large-scale cloud storage systems. The primary goal of the proposed architecture is to enhance the scalability of namespace directory in large-scale file systems. Structural shift from distinguished distributed model to interconnected distributed model and conducting effective coordination among file servers for namespace management are two key solutions considered in the context of proposed architecture. To this intent, a coordination protocol is designed for communication among file servers, and maintaining user transparency in the presence of different file system actions/reactions. The experimental results, obtained via emulations under different network conditions and cloud storage sizes, show up to 43.9% availability and 37.8% connection throughput improvements with negligible storage overhead compared to the latest released version of Hadoop distributed file system.

Keywords Scalability · Availability · Cloud storage systems · Hadoop distributed file system · Namespace directory

1 Introduction

Emerging adoption of cloud computing in different aspects of information technology such as financial services, social networks, e-health, media and entertainment is driving the growth demand for cloud storage systems [1]. As content is created

✉ M. Maghsoudloo
mo.maghsoudloo@gu.ac.ir

N. Khoshavi
nkhoshavinajafabadi@floridapoly.edu

[1] Department of Computer Engineering, Faculty of Engineering, Golestan University, Gorgan, Iran

[2] Department of Computer Science and Department of Electrical and Computer Engineering, Florida Polytechnic University, Lakeland, USA

anytime and anywhere on billions of end systems, cloud storage infrastructure is needed to store, manage and retrieve massive amounts of data [1, 2]. Cloud storage market is projected to reach a total market size of $92.488 billion by 2022 with respect to compound annual growth rate of 53% from 2011 to 2016 [3].

A cloud storage system provides a means of permanent storage of set of data in the form of files in which clients can access files via a lightweight user agent [1–3]. Unlike the traditional local file storages, clients and storage resources can be dispersed over the Internet [1]. Client-side machine splits files into constant-size objects that are stored on different storage resources. This type of distribution is transparent from client viewpoint so that it gives an impression that the objects are present at the same geographical location [3].

Google file system (GFS) and Hadoop distributed file system (HDFS) are two of the most commonly used cloud storage systems for dealing with huge clusters [3]. GFS and HDFS are similar in many aspects, especially in architecture. GFS contains single master node (master) and multiple chunk servers (slaves). HDFS contains single NameNode (master) and many DataNodes (slaves). In HDFS, chunk location information is consistently maintained by NameNode, while GFS's master simply checks the status of the chunk servers for their information at startup [3]. Since the Hadoop is an open-source software framework, most research works and innovative ideas have been applied on the HDFS architecture.

For shared resource management, the centralized approach has two major drawbacks and centralized NameNode is no exception: the single point of failure (SPOF) problem and performance bottleneck. As HDFS architectures depend on a single master node which at times proves to be a SPOF. If it goes down, the slaves will lose control for blocks [4–8]. Moreover, since all metadata are stored in the NameNode, client requests to an HDFS cluster must first pass through it [9]. However, the rapid growth of businesses (such as Uber [10]) made it difficult to scale reliably without slowing down data analysis for thousands of users who are making millions of queries each week [10]. Therefore, an effective solution to overcome these challenges is structural shift from centralized to distributed model [9, 11–15].

The latest released version of Hadoop, called HDFS Federation [9], is conducted based on the above issue. It tries to overcome the scalability limitations of previous versions by adding support for multiple NameNodes to HDFS. HDFS Federation improves the existing HDFS architecture through separation of namespace and storage [9]. All NameNodes work independently and do not require any coordination with each other. Each NameNode manages a separated part of namespace called block pool. DataNodes store blocks for all the block pools in the cluster. Consequently, a NameNode failure does not prevent the DataNodes from serving other NameNodes in the cluster [9]. Adding more NameNodes to the cluster scales the file system read/write throughput and can overcome the performance bottleneck problem.

However, the experimental results, presented by this paper, reveal that the un-connected self-ruled NameNodes and static namespace partitioning faces in the HDFS Federation two serious problems: First, the SPOF problem due to lack of coordination between independent NameNodes deprives some users of the consistent data layout; second, limitation of system flexibility in dealing with network dynamics

such as user mobility and migrations, unstable network conditions (node failures and unpredictable Internet traffic behavior) and unbalanced loads. The experimental results show that the negative effects of both above issues become more complicated during scaling up the system (increasing number of clients, network size and utilization).

In order to enhance the scalability/availability of large-scale cloud storages, this paper presents Elastic HDFS, an interconnected distributed architecture for HDFS implementation. In the proposed architecture, NameNode is split into slices and geographically distributed among chunk servers. In contrast to HDFS Federation, chunk servers are coordinated and a coordination protocol is designed for communication among chunk servers. A slice of NameNode, associated with a DataNode, contains a certain part of metadata information of files' objects. In fact, each NameNode is the Parent of some client machines which are geographically located in particular area. Client machines communicate with their Parents for all actions, such as upload, read and edit files, and chunk servers communicate with each other to commit jobs. In the case of chunk server failure, unstable network conditions and user mobility, coordination of chunk servers helps clients keep accesses to slaves without significant disruption. In the absence of active parent, other chunks can issue the client requests during failover mechanism. Concerning the defined functions and messages of coordination protocol, all of the chunks are able to redirect the requests to the ones, which are responsible of them (Parents). Moreover, on the congested networks, the coordination among different chunk servers conducted by Elastic HDFS can help distribute the traffic over the network in the favor of overloaded chunks. In a general view, the cooperation among chunk servers with different roles improves the accessibility of data stored by the DataNodes in the presence of unpleasant events.

In order to compare the effects of design decisions, different network infrastructures contain 16, 32 and 64 subnets which are emulated via Graphical Network Simulator 3 (GNS3) [16]. The Intelligent Java IDE (IntelliJ IDEA) [17] is used for building modified java classes classified into Hadoop packages. To measure the impact of the HDFS architectures on the performance and availability of Hadoop, the standard TestDFSIO, TeraGen and TeraSort benchmarks [18] are used. The primary metric used to measure the performance of HDFS architecture is throughput, and availability is measured considering the factors causing interruption in clients' accesses. In order to evaluate the robustness of HDFS architectures against network congestion, random failures and clients' movements, the throughput is investigated in the presence of fake secondary traffic, different numbers of chunk server failures and client migrations. The experimental results reveal that Elastic HDFS causes that the descending slopes of throughput decrease by 32.6%, 77.4% and 42.1% during growth of congestion ratio, failure ratio and migration ratio. The maximum availability improvements of the proposed architecture during the mentioned scenarios are about the 43.9%, 25.8% and 22.9% for the networks with different congestion, migration and failure ratios.

Coordination among HDFS chunks conducted by Elastic HDFS enables other chunks to deal with client requests in the absence of responsible ones due to high traffic, failure or client migration. Although the coordination among HDFS chunks established by Elastic HDFS increases the processing efforts on the server side, it

can help reduce the dependency of overall server-side operations on the specified NameNode.

The rest of the paper is organized as follows: Sect. 2 describes the base Hadoop distributed file system. Section 3 explains the problem and structures of the related work. The features of proposed HDFS architecture and HDFS coordination protocol are proposed in Sects. 4 and 5. Section 6 shows the result analysis. Finally, Sect. 7 concludes the paper.

2 Hadoop distributed file system

The Hadoop distributed file system (HDFS) is a distributed file system built to run on commercial of the shelf hardware [3]. The main goal of HDFS is to use commonly commodity servers in a large cluster, where each server has a set of inexpensive disk drives. It is originally implemented using the Java language for the Apache Nutch Web search engine [3]. HDFS provides high-throughput access to application data and is suitable for applications that have large datasets [3].

Figure 1 shows the base architecture of HDFS containing four major software components: Client Machine, NameNode, DataNodes and Secondary NameNode. Client machines have Hadoop installed with all the cluster settings, but are neither a master or a slave. Instead, the role of the Client Machine is to load data into the cluster via establishing a connection to a configurable TCP port on the NameNode [19]. An HDFS cluster consists of a single NameNode that manages the file system namespace, and number of DataNodes that manages storage attached to the clusters [19]. The NameNode executes file system namespace operations like opening, closing and renaming files. It also determines the mapping of blocks to DataNodes [19]. The DataNodes perform object creation, deletion and replication under the

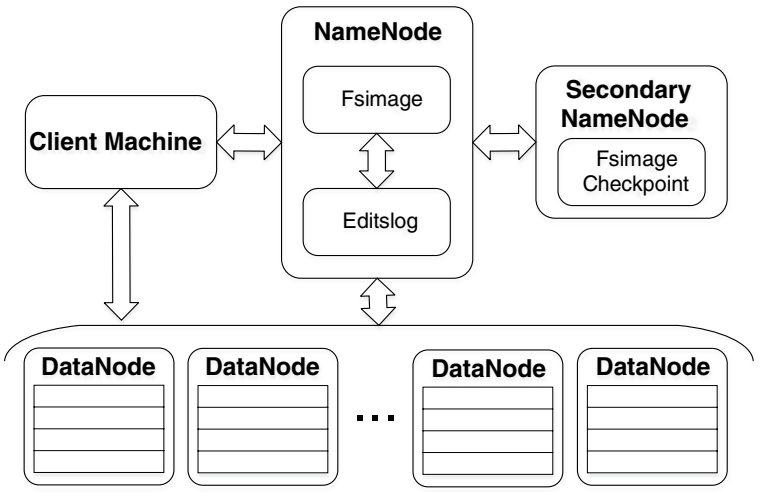

Fig. 1 Architecture of base distributed file system of Hadoop

management of NameNode [19]. The existence of a single NameNode in a cluster significantly simplifies the architecture of the system [19]. The NameNode is the arbitrator and repository for all HDFS metadata [19].

The NameNode stores the metadata of the HDFS. The state of HDFS is stored in a file called fsimage and is the base of the metadata. During the runtime, modifications are just written to a log file called editslog. On the next startup of the NameNode, the state is read from fsimage. Then, the changes from editslog are applied to that and the new state is written back to fsimage. After this, editslog is cleared and is now ready for new log entries [19].

NameNode restarts are rare which means editslogs can grow very large for the clusters where NameNode runs for a long period of time. Moreover, in the case of crash, we will lose huge amount of metadata since fsimage is very old. Therefore, to overcome this issue, we need a mechanism which will help us reduce the editslog size containing the most recent fsimage leading to load reduction on NameNode [20]. The Secondary NameNode helps to overcome the above issues by taking over the responsibility of merging editslog with fsimage from NameNode. The secondary NameNode merges fsimage and editslog periodically and restricts the size of editslog while being typically run on a different machine than the primary NameNode. It receives the editslogs from NameNode in uniform intervals and applies them to fsimage. Once it has new fsimage, it copies back to NameNode. NameNode will use this fsimage for the next restart, which will reduce the startup time. The main idea behind establishing a secondary NameNode is to equip the HDFS with checkpoint system. It is just a helper node for NameNode, and in the case of NameNode failure, it will not be employed as a replacement or backup node [20].

3 The problem statement and related work

One of the most important aspects of any cloud computing solution is the availability of the cloud [4]. Availability refers to the uptime of a system, a network of systems, hardware and software that collectively provide a service during its usage [4–6]. When a cloud solution is not available, it practically does not exist. In effect, any data or apps that are accessed via the cloud cannot be utilized as long as the cloud server remains unavailable. Master failure and scalability limitation of conventional HDFS architecture are two factors that can affect the availability of HDFS as follows:

- For any un-planned events such as node failure, the entire cluster is not available until the NameNode is brought up. For planned events such as hardware or software upgrades on NameNode, it would also result in the cluster unavailability [20].
- Single NameNode causes a bottleneck of performance since all the access requests to the file system have to contact the NameNode [19]. A NameNode is considered available if it is up and it meets a service-level objective that can be affected by poor performance [19].

In Hadoop 1.x, each cluster has a single NameNode, and if that machine fails, the whole cluster will not be available [19]. In Hadoop 2.x, HDFS feature addresses the above problem, by providing an option to run two NameNodes in the same cluster in an active/passive configuration with a hot standby that are completely synchronized through the shared storage [20]. However, this causes a new SPOF problem because the shared storage is a single point which stores all edit logs [20]. The next versions of Hadoop 2.x also have two NameNodes that are configured and synchronized at all times [21]. Nevertheless, they implement Quorum Journal Manager (QJM), instead of the shared storage. Using QJM to maintain consistency of active and standby state requires that both nodes be able to communicate with a group of JournalNodes (JNs) [21]. When the active node modifies the namespace, it logs a record of the change to a majority of JournalNodes. The Standby NameNode watches the JNs for changes to the edit log and applies them to its own namespace [21]. Although the QJM can overcome the SPOF problem of shared editslog in the former versions of HDFS, two challenges in achieving high availability of scaled-up HDFS still persist: (1) performance bottleneck problem and (2) multiple-failure problem [22].

The main scalability issue of HDFS is the performance and throughput of the NameNode, the directory tree of all files in the system that tracks where data files are stored. Since all metadata are stored in the NameNode, client requests to an HDFS cluster must first pass through it [9]. HDFS was designed as a scalable distributed file system to support thousands of nodes within a single cluster. With enough hardware, scaling to over 100 petabytes of raw storage capacity in one cluster can be easily achieved by HDFS.

Furthermore, a StandBy sparing system with two modules can tolerate just one machine failure. In the event of failure on both Active and StandBy NameNodes, the availability of HDFS can be compromised again. Considering the fact that availability is ultimately the holy grail of the cloud storage systems, the StandBy sparing cannot be enough to minimize the risk of NameNode outages. While the volume of data in HDFS has been growing exponentially [6], the importance of NameNode reliability also increases rapidly. Several methods have been reported in the literature to enhance the NameNode reliability. Most of them are based on using N-Way Replication mechanisms for metadata replication (such as AVATOR Node [23], NCluster [24] and KARMA [2]) or applying erasure codes or blockchain mechanism on metadata for error detection/correction (such as MICS [25] and ASSER [26]). The basis of ideas used by the aforementioned techniques is to use data redundancy for NameNode reliability improvement. On very large clusters with many files, NameNode memory becomes the limiting factor for scaling [11]. Thus, NameNode will not be able to handle the extra data which makes the scalability issue more complicated [11]. Even though these techniques enhance NameNode reliability, they also impose significant bandwidth, performance, energy and area overheads [12].

A limited study has been conducted on the pros and cons of scaling out NameNode architecture [11–15]. For instance, the focus of [12] is on the namespace partitioning among a cluster of NameNodes. Moreover, replicas of each fragment are dispersed among the clusters. Both of namespace partitioning and the locations of replicas can be changed dynamically under the management of a watchdog node, called ZooKeeper [12]. While ZooKeeper adjusts cooperation

among NameNodes and keeps them synchronized, it can also be a new single point of system failure and potential performance bottleneck.

Finally, Apache Software Foundation triggers to propose a new version of HDFS to overcome the scalability limitations of previous versions by adding support for multiple NameNodes. HDFS Federation (Hadoop 2.9) [9] improves the existing HDFS architecture through a clear separation of namespace and storage [9]. It enables support for multiple namespaces in the cluster to improve scalability. Figure 2 illustrates the architecture of HDFS Federation. As shown in Fig. 2, NameNodes are federated indicating that all these NameNodes work independently and do not require any coordination with each other. Each DataNode registers with all NameNodes in the cluster. DataNodes send periodic heartbeats and handle commands from NameNodes. Each NameNode manages a separated part of namespace called block pool. Therefore, a block pool is a set of blocks that belong to a single namespace. DataNodes store blocks for all the block pools in the cluster. Consequently, a NameNode failure does not prevent DataNodes from serving other NameNodes in the cluster.

Adding more NameNodes to the cluster scales the file system read/write throughput and can overcome the performance bottleneck problem of architecture with single NameNode. However, the experimental results explained in Sect. 6 show that the un-connected self-ruled NameNodes and static namespace partitioning results in two serious problems in HDFS Federation:

- While multiple NameNodes are being used to manage all DataNodes, failure of one NameNode cannot affect the data of other ones. However, access of some users correlated with the failed namespace can still be affected. The failed namespace is unavailable until the corresponding namenode comes back to the

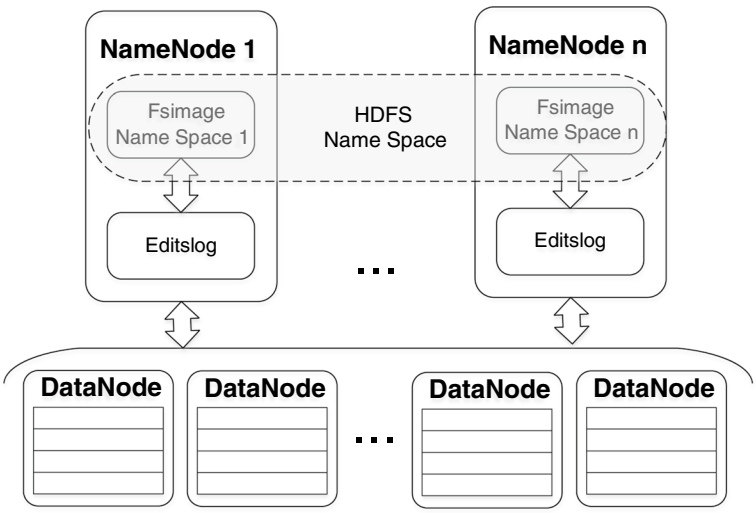

Fig. 2 Architecture of HDFS federation

cluster. Thus, the SPOF problem due to lack of coordination between independent NameNodes deprives some users of the consistent data layout.

- In HDFS Federation, the geographically closest NameNode serves the requests of the clients resided in a specific area. Even though the responder NameNode is physically closest to the client machine, the NameNode that is farther away from the clients may lead to better response delay. User mobility and migrations, unstable network conditions (node failures and unpredictable Internet traffic behavior) and unbalanced load on the federated independent NameNodes cause that the closest NameNode might not continuously be the best choice for the user to interact with the file system.

The experimental results (Sect. 6) show that the negative effects of both above issues become more complicated during scaling up the system (increasing number of clients, network size and utilization).

4 Proposed distributed architecture for NameNode availability enhancement

The centralized model for NameNode implementation is the main cause of scalability challenges and the major barrier to expand metadata directory of HDFS. NameNode is the key resource of HDFS, shared by client machines as a directory for file objects. For shared resource management, the centralized architectures cannot expand effectively.

To address this issue, we propose an interconnected distributed architecture for metadata directory of HDFS as an alternative solution for federated NameNode. The key innovation of the proposed architecture is to keep metadata information along with the file objects in chunk servers. Figure 3 illustrates how distribution can be

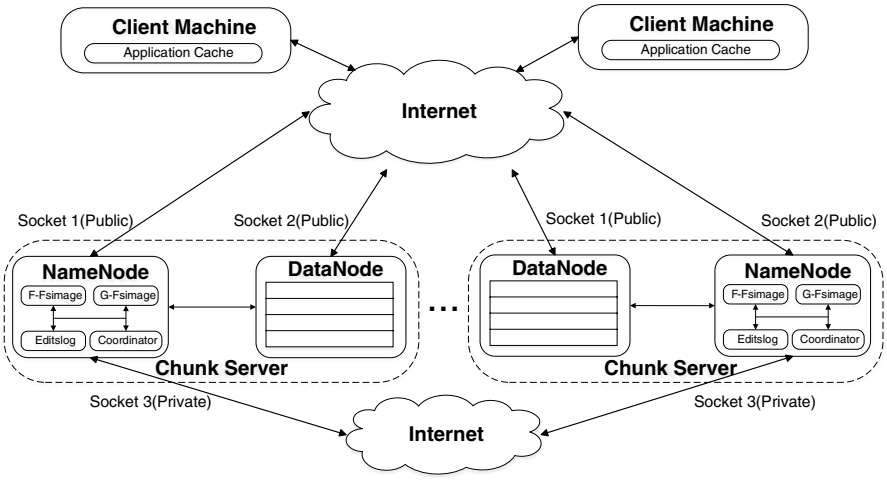

Fig. 3 Proposed distributed architecture for namenode of HDFS

applied on the HDFS architecture. As shown in Fig. 3, the metadata namespace is split into fixed-size slices and geographically distributed among chunk servers. Each chunk server acts like a virtual machine (VM) that can run along with other VMs on a same physical machine (PM). A chunk server consists of a DataNode associated with a slice of directory. Each slice contains a certain part of metadata of NameNode. Despite HDFS Federation, NameNode slices communicate to each other for metadata replication, management and migration. Different slices of NameNode work together under the management of new software module (NameNode Coordinator) via a pre-designed coordination protocol. Coordination protocol involves messages, actions and reactions for NameNodes collaboration. During normal operation, each client machine contacts to a slice of NameNode with the help of anycast networking method, and different slices collaborate with each other to commit the requests of client machines. With anycast, a single IP address is assigned to the network adapter interfaces of chunks that are bound to the ports (50070 and 50470) listened by NameNode modules. Anycasting is supported explicitly in IPv6 and by the BGP (Border Gateway Protocol) in IPv4. By anycast networking method, the same IP prefix is advertised from multiple NameNodes. Anycast is a network routing and addressing mechanism which enables multiple topologically diverse servers in the Internet to share the same IP address to provide service. The shared IP address is called an anycast address. User requests sent to the anycast address are routed to the topologically nearest server. This helps cut down on latency and bandwidth costs, improves load time for users and improves availability. It is important to remember that topographically closer does not inherently mean geographically closer, though this is often the case. Anycast is linked with the BGP protocol, which ensures that all of a router's neighbors are aware of the networks that can be reached through that router and the topographical distance to those networks. The main principle of anycast is that an IP address range is advertised in the BGP messages of multiple routers. As this propagates across the Internet, routers become aware of which of their neighbors provides the shortest topographical path to the advertised IP address. The responder NameNode may not be the closest physically to client machine, however, the one 'network-close' to it. Two distinguished public TCP sockets are established on the chunk servers for NameNode and DataNode connections. Regarding Fig. 3, Socket 1 and Socket 2 are the endpoints of two-way communications for client machines with NameNode and DataNode, respectively. Moreover, a private TCP socket (Socket 3 in Fig. 3) is taken into account for message passing between NameNodes in the context of coordination protocol.

In the proposed architecture, NameNodes can take two different roles:

- *Parent* Each NameNode is the Parent of files uploaded by caller client machines (which is possibly located in the nearby area of chunk). An image file (F-fsimage in Fig. 3) is embedded in the structure of the proposed NameNode to keep the addresses of DataNodes (sharer tags), which is the host of its children's objects. All actions requested by a client, such as upload, read and edit files, should be committed by the Parent. However, it is not necessary for the client machines to perform all actions on a file just via establishing a direct connection to Parent of the files.

- *Grandparent* Each NameNode can also be a Grandparent of some files. In fact, chunks can store the objects of files that are children of the other NameNodes. Grandparents are not permitted to serve the requests of client machines for actions on their Grandchildren. They just perform actions committed by the Parents. Grandparents' DataNodes store the replicas of objects sent by their Parents. In order to store addresses of them (owner tags), a separated image file (G-fsimage in Fig. 3) is considered in the proposed architecture of NameNode.

In the following, different scenarios are considered to clarify the interactions among client machines, Parents and Grandparents in the context of proposed architecture.

4.1 File uploading/creating

Figure 4 shows the file uploading mechanism intended in the proposed HDFS architecture. In order to upload files to enhanced HDFS, the client machines act as an agent for users. Client machine receives file X and splits it into fixed-size objects

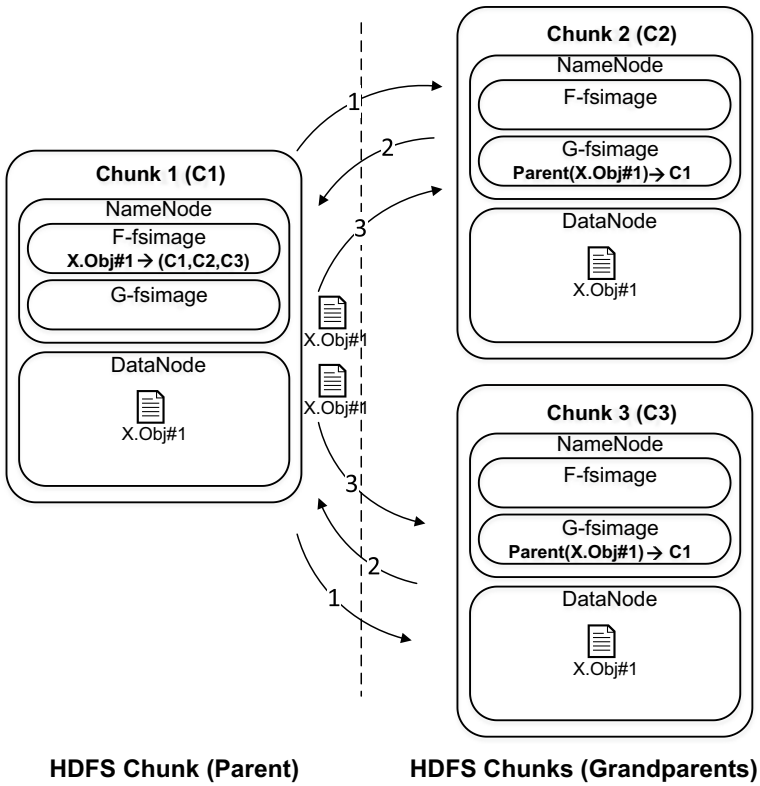

Fig. 4 Proposed file uploading/creating mechanism

(X.Obj#1, X.Obj#2 and X.Obj#3). The client requests are navigated to the most optimal and closest chunks based on the anycast mechanism and BGP routing protocol. When the request reaches the server (C1), it is passed to the TCP socket connected to NameNode module as its Parent. Despite the basic file transfer mechanism of HDFS, the responsibilities of client machine end at this time. After transferring objects, Parent stores the objects and makes two copies of them. To locate copies in the file system, the Parent begins to communicate with other NameNodes considering greater priority for closer chunks (transmission 1). Given the amount of unused storage of chunks, they can accept or reject the requests of Parent for receiving copies (transmission 2). In the case of acceptance, the copies are sent to the acceptor chunks along with the Parent Acknowledgement (transmission 3). From this moment onward, the acceptor chunks (C2 and C3) play the role of Grandparent to mentioned file. Finally, F-fsimage of Parent and G-fsimage of Grandparents are updated via adding Grandparent addresses in the set of objects holder in F-fsimage and inserting Parent address of stored object in G-fsimage of Grandparents. By means of the proposed mechanism, at least three NameNode slices (one Parent and two Grandparents) keep the metadata information of each object, stored in DataNodes.

4.2 File editing/deleting

Hadoop includes various shell-like commands that directly interact with HDFS and other file systems that Hadoop supports. These commands support most of the normal files system operations like copying files, changing file permissions and so on. It also supports a few HDFS-specific operations like changing replication of files. Editing files at the client side is not supported in basic HDFS, as it works on the principle of write once/read many [2]. However, nowadays, files can be remotely edited using a lightweight Web server that permits clients to use Hadoop directly from the browser [3]. In the context of the proposed distributed HDFS, Parent sends any updates on files to Grandparents after client machine operations. Moreover, Parent grants permission of specific editing on files to Grandparents, allowing them to apply updates on the replicas. Permissions will be retrieved after acknowledgement of Grandparent or after passing a specified deadline. While the fsimage file format is very efficient to read, it is unsuitable for making small incremental updates like renaming a single file. Thus, rather than writing a new fsimage every time the namespace is modified, the NameNode instead records the modifying operation in the editslog for durability [19]. Figure 5 implies the proposed update propagation mechanism. After each small modification on files, Parent sends updates on editslog to Grandparents along with its corresponding permissions with a specified time-out duration. To handle major updates on fsimage, replication-pipelining method can be effective. When a client is writing data to an HDFS file, its data are first written to the main copy in the Parent's DataNode. Suppose the HDFS file has a replication factor of three. The Parent (C1) then flushes the data block to the Grandparents (C2 and C3) with anycast (transmission 1) and locks the file until update operation finishes. The first Grandparent (C2) starts receiving the data in small portions, writes each portion to its local repository and transfers that portion to the second

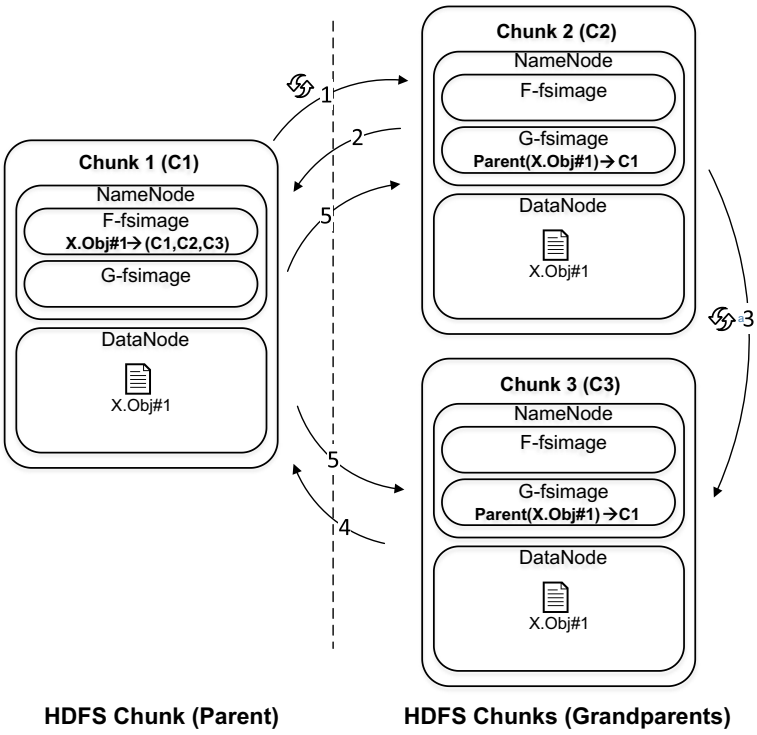

Fig. 5 Proposed file editing/deleting mechanism

Grandparent (C3). After applying updates on the objects, Grandparents inform the Parent via an update acknowledgement message considered in the message set of the proposed coordination protocol (transmission 2 and transmission 4). If for any reason different coordination messages are lost during the update propagation mechanism or applying updates on the objects is not performed completely, the Parent will not commit the update operation. When Parent receives all the acknowledgement messages, the update propagation mechanism is completed, the file is unlocked and commitment messages are sent to the Grandparents (transmission 5).

4.3 Parent discovery

In order to serve the client requests for different file actions, all of the requests should be received by the Parent. Figure 6 illustrates the Parent discovery process. All of the requests, issued by the client machines to satisfy the user's demands for file uploading/editing, are sent with the help of anycast addressing mechanism. In the other words, this type of messages is just routed to one chunk server, which is the network-closest one, and requests will not be replicated and routed to several chunk servers in underlying network (as intended in the context of multicast and broadcast addressing mechanisms). Therefore, the requests issued and sent by the

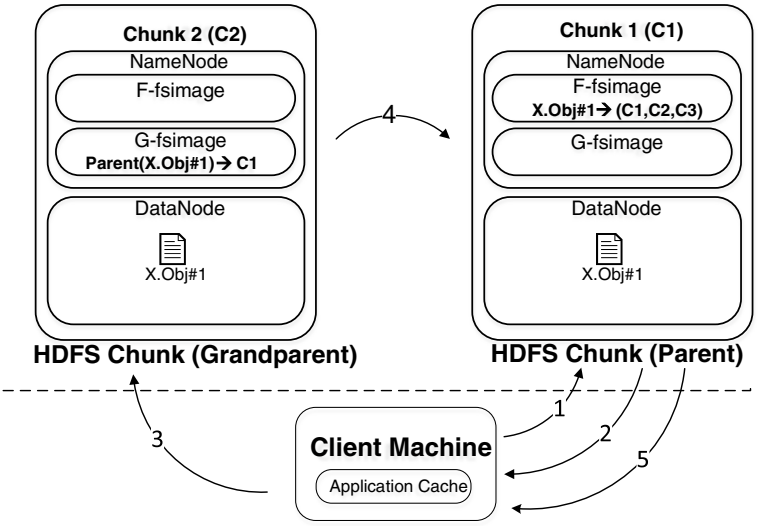

Fig. 6 Proposed Parent discovery mechanism

clients can be three types of recipients: the Parent, one of the Grandparents or one of the other chunks that have not taken any role for the file.

The first scenario The client request directly reaches the Parent of the file (transmission 1). Based on the anycast mechanism, the requests are always navigated to the closest chunk. As long as there are no topology changes in the network, a stationary client can keep its connection to HDFS via a unique chunk. In this case, the Parent of different files of a client can be a same NameNode. Consequently, the Parent is directly involved in request handling and response preparation mechanisms (transmission 2).

The second scenario The client request reaches one of the Grandparents of the file (transmission 3). As Grandparent stores address of the objects' Parent in G-fsimage, it can redirect the request to the Parent (transmission 4) via reverse proxy mechanisms such as Application Request Routing (ARR) [27] or Nginx [28]. The reverse proxy server takes requests from the clients and forwards them to one of the servers. This issue is taken into account transparently from client point of view. Grandparent analyzes incoming request and delivers it to the right chunk (Parent). Grandparent establishes an HTTP connection to Parent through private TCP sockets of NameNodes. Grandparent caches client requests and redirects them to the Parent. Then, Parent delivers client's response directly (transmission 5). This condition arises due to client moving over the Internet or Parent connectivity problems. If the problem persists, it can be concluded that the Parent might no longer the closest chunk to the client. The circumstances leading to metadata migration are explained in the rest of this section.

The third scenario The client request reaches the other chunks that have not taken any role for the file. Then, the request is redirected to all private sockets of NameNodes via multicasting to ensure that request reaches the Parent. Finally, the Parent can individually handle the request exactly in the same way explained in the previous scenario.

An experiment was conducted to extract the probabilities of three above scenarios. A network infrastructure with 32 different subnets (under the organization of different ISPs), 32 client machines and chunk servers (resided in different subnets) was emulated via emulator. More details about the emulation environment is mentioned in Sect. 6. Moreover, 30 GB data are fairly generated by client machines spread out over the emulated network. In this case, on average about 18.31% of overall links' bandwidth was used for benchmarks. The results show that the Parent, one of the Grandparents and other chunk servers, respectively, receives about 42.11%, 25.42% and 32.47% of client's requests. As the results under normal conditions of the network, most of the client requests reach the Parent (C1) and are not required to be redirected to the other chunk.

4.4 Parent/Grandparent failure

Distribution of metadata among chunks and using a separated file (G-fsimage) for storing owner tags of each object in NameNodes lead to increased reliability of HDFS. Because of replication factor of three for storing objects in different chunks' DataNodes, metadata of a specific object are stored in exactly one F-fsimage of its Parent (in the format of sharer tag) and in two G-fsimage of its Grandparent (in the format of owner tag). Consequently, chunk failures can be tolerated via chunks cooperation.

Figure 7 shows how the proposed architecture can tolerate Parent failure. In the case of Parent failure, anycast mechanism routes the client requests to the other closest chunk (transition 1). While Grandparents contain metadata of the requested file along with its objects (in the associated DataNode), Parent failure can be hidden from user's point of view via Grandparents' interactions. If one of the Grandparents receives the requests, it tries to establish a connection to Parent through private TCP sockets. Due to the failure, the Parent cannot reply to the request, and the connection time-out error occurs. Then, the Grandparent, who has received the request, informs the other chunks of failing one of NameNodes (transition 2). After that, it redirects the client request to all private sockets of NameNodes via multicasting (transition 3). Grandparents' DataNodes store the replicas of objects and store addresses of Parent in G-fsimage of their NameNodes. Therefore, Grandparents can issue the client requests in the absence of active Parents and during failover mechanism. If the client request reaches the other chunks that have not taken any role for the file, the request is redirected to all private sockets of NameNodes under management of Parent recovery mechanism. Therefore, the request reaches the Grandparents and can be handled by them. Subsequently, Grandparents reply to the client's request on behalf of Parent (transitions 4).

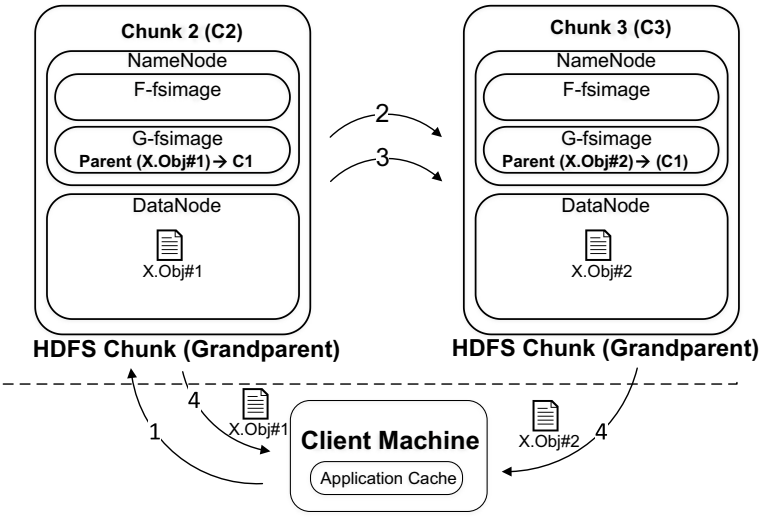

Fig. 7 Proposed Parent/Grandparent failure handling mechanism

In the case of Grandparent failure, Parent can be aware of this failure during interactions with Grandparents intended in the context of mentioned scenarios. Eliminating one of owner tags can be tolerable while the main sharer tag, kept by the Parent, and the other owner tag, kept by the second Grandparent, are still alive. After failure detection, Parent begins to communicate with other NameNodes for finding new Grandparent for the file, as same as file uploading mechanism.

In the case of both Parent and Grandparent(s) failures, the first reaction of the proposed architecture is same as its reaction when just the Parent fails. Anycast mechanism routes the client requests to the other closest chunk, and the remaining Grandparent(s) finally receives the request and informs the other chunks of failure occurrence. After that, it redirects the client request to all private sockets of NameNodes. As long as at least one Grandparent remains, they can issue the client requests in the absence of active Parents and during failover mechanism.

In the experimental evaluation, the throughput of connections between client machines and chunk servers managed by HDFS Federation and Elastic HDFS in terms of different numbers of subnets/clients and different failure ratios of chunks is assessed. To investigate the effects of Parent and Grandparent(s) failures on the throughput more deeply, the amount of throughput reduction is extracted from the experiments with respect to the above scenarios for the network with 16 subnets/clients. In the worst case, the throughput of connection, established for accessing a file, is degraded about 14.7%, 18.9% and 27.0% in the case of a Parent, a Parent and a Grandparent and a Parent and two Grandparents failures, respectively. It should be noted that throughput degradation increases as network scale grows to 32 and 64 subnets, due to increased overhead of anycasting and multicasting mechanisms issued during failover period.

4.5 Long-term migration of users

In order to enable user interaction with the proposed file system, the client machine should make a connection with the chunks, taken the role of Parent or Grandparent for the user's files. In the first contact of the user, the closest HDFS chunk accepts the responsibility of being Parent of some user's files in the system. The Parent will be the network-closest chunk to client, as long as the network conditions are normal and stationary user does not have any significant movement over the network. In this case, every time the user tries to access or work with his/her files in the same geographical area where the files have been uploaded for the first time, the Parent is the best choice for the user. Figure 8 illustrates the situation where the user migrates to another geographical zone through the management of different Internet service providers (ISPs), so that the Parent is not the network-closest chunk to the client machine. After the migration, communicating with the previous Parent can impose some overheads in terms of access delay that can lead to degraded system availability. Therefore, in the event of long-term migration of a user, a new Parent should be selected among new close chunks for storing objects and metadata of heavily accessed file. While the location of keeping files and metadata may affect the data integrity, this process should be handled with the awareness of the user. As shown in Fig. 8, with user permissions, the network-closest chunk is selected as the new Parent, and the new Parent communicates with its neighbor for creating new set of Grandparents. User grants permission to the new Parent, as a third-party auditor (TPA), for downloading the file's objects from previous Parent. The TPA is an entity, which has all the necessary expertise and capabilities required to act on behalf of the client [29]. After objects migration, sharer tag/owner tags are created and entered in the F-fsimage/G-fsimage of Parent/Grandparent. Finally, the client withdraws the permissions of previous Parent and Grandparents for the mentioned file and allocated spaces of NameNodes and DataNodes are released for new objects and metadata.

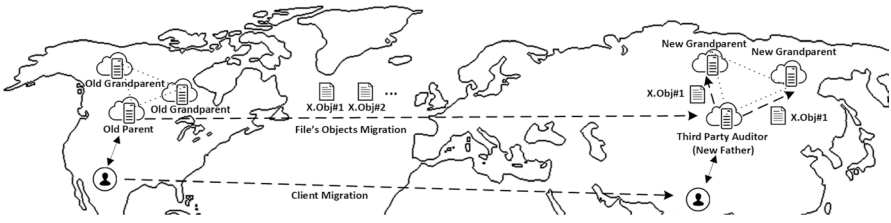

Fig. 8 Proposed client migration handling mechanism

5 NameNode coordination protocol

In order to keep distributed directory coherent, a coordination protocol is employed as the common communication language of different slices. Parent and Grandparents should communicate with each other to commit the requests so that the uniform logical view of the storage system is kept for the client.

Table 1 and Fig. 9 show the messages and interactions among different slices of metadata directory in the context of the proposed coordination protocol. The source and destination of each message, types of sockets and addressing method are mentioned in Table 1 along with a brief description of messages. The numbers, assigned to each coordination message in Table 1, have been specified via labels on the transitions in Fig. 9. As illustrated by Fig. 9, the clients and HDFS chunks (with different roles) can issue a coordination message. They can set up a connection to the private or public sockets of each other for message passing. An issued message can be sent to the destination over the created connection considering three types of communication: unicast, multicast and anycast.

The client machines send the clients' requests (for file locating and processing) through establishing connections to the public sockets of HDFS chunks. The routing protocol (with the help of anycast mechanism) delivers the client request to the network-closest HDFS chunk. After receiving a request from a client, the HDFS chunks collaborate to commit the client's request. The messages, intended for HDFS chunks communication, are exchanged over connections established between private sockets of HDFS slices. Different multicast groups are taken into account to facilitate communication of HDFS chunks, especially between Parent and Grandparents. All the HDFS chunks are added into a multicast group. Moreover, a number of separated multicast groups are also considered for Parents and Grandparents communication. In the case of different events (such as chunks failure, client migration and load balancing), Parents and Grandparents try to keep almost three replicas for each object in the network-closest chunks to the client, according to their duties.

6 Evaluation results

In order to validate the ideas and compare the effects of proposed architecture with earlier ones, emulation-based verification method is exploited. Emulation-based verification is the process of imitating the behavior of a system under design with the help of another real system (machine).

6.1 Experimental setup

To evaluate design decisions under variable network conditions, different network infrastructures are emulated via Graphical Network Simulator 3 (GNS3) [16]. GNS3 is an open-source software used to emulate, configure, test and troubleshoot virtual and real networks [16]. GNS3 consists of two software components:

Table 1 Messages of namenode coordination protocol

Number	Request	Source	Destination	Socket	Addressing	Description
1	File locating/processing	Client	All chunks	Public	Anycast	Messages issued by the client machines to satisfy the user's demands for file uploading/editing
2	Parent offer	A chunk	Client	Public	Unicast	Messages issued by a HDFS chunk to inform the client that is ready to be the Parent of his/her file
3	Replica placement	Parent	All chunks	Private	Multicast	Messages issued by the Parents for locating replicas of objects in the other HDFS chunk servers
4	Admission	Some chunks	Parent	Private	Unicast	Messages issued by the potential Grandparents to admit requests of Parent for storing objects
5	Rejection	Some chunks	Parent	Private	Unicast	Messages issued by the HDFS chunks to reject the request of Parent for storing objects due to heavy load/low space
6	ACK	Parent	Grandparent	Private	Unicast	Messages issued by the Parents to confirm the role of Grandparent for HDFS chunks after receiving admission
7	NACK	Parent	Some chunks	Private	Unicast	Messages issued by the Parents to refuse the role of Grandparent for HDFS chunks after receiving admission
8	File processing permissions	Parent	Grandparents	Private	Multicast	Messages issued by the Parents to grant permission of the file processing to the Grandparents
9	File processing ACK	Grandparent	Parent	Private	Unicast	Messages issued by the Grandparents to show the confirmation of enforcing permissions on access list's rules
10	File processing commitment	Parent	Grandparents	Private	Multicast	Messages issued by the Grandparents to inform the Parent of fulfilling client request for file editing
11	Request redirect#1	Grandparent	Parent	Private	Unicast	Messages issued by the Grandparents to redirect client's request to the Parent via Parent discovery mechanism
12	Request redirect#2	A chunk	All chunks	Private	Multicast	Messages issued by the HDFS chunk servers to find Parent/grandparent for Parent discovery/failure handling
13	Failure alert	Grandparent	All chunks	Private	Multicast	Messages issued by the Grandparents to find the other Grandparents of a file in the case of Parent failure
14	Auditing request	A chunk	Client	Public	Unicast	Messages issued by the HDFS chunk servers to request of granting auditing permissions from client

Table 1 (continued)

Number	Request	Source	Destination	Socket	Addressing	Description
15	Auditing permission	Client	Chunk	Public	Unicast	Messages issued by the client to affirm TPA role of a HDFS chunk server for metadata migration
16	Permission withdrawal	Client	TPA/parent	Public	Unicast	Messages issued by the client to deny TPA/parent role of a HDFS chunk server for metadata migration
17	Object migration request	TPA	Parent	Private	Unicast	Messages issued by the TPA to request of sending metadata of a specific file from its Parent
18	Grandparent withdrawal	Parent	Grandparents	Private	Multicast	Messages issued by the Parent to take the role of Grandparent back because of metadata migration or load balancing

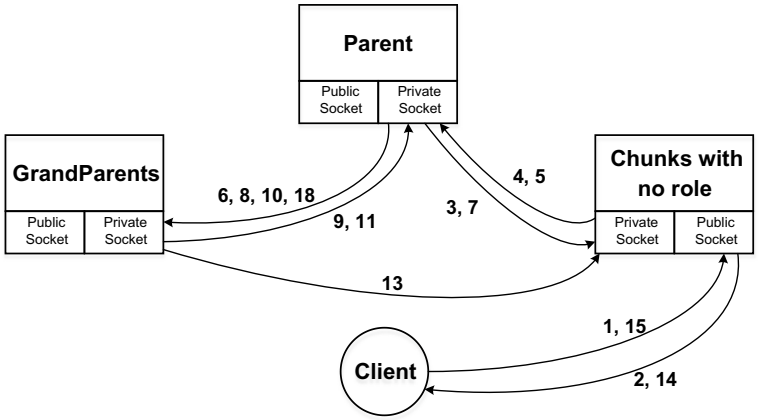

Fig. 9 Sequence of messages in the structure of coordination protocol

the GNS3 Graphical User Interface (GUI) and the GNS3 virtual machine (VM). GNS3 GUI is the client part of GNS3 and used to create network topologies consisting of virtual machines and network components.

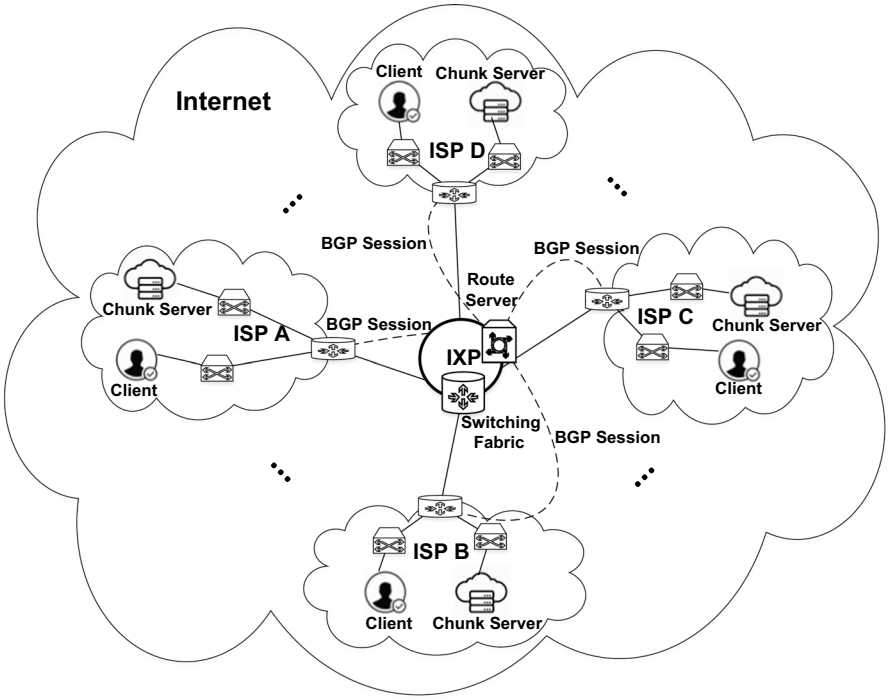

Fig. 10 Emulated network topology and infrastructure

Figure 10 illustrates the topology of emulated network. To evaluate the design decisions, 16, 32 and 64 different subnets are emulated under the organization of separated Internet service providers (ISPs). One client machine and one HDFS chunk server are put into each subnet. A PC and a physical machine (PM) are intended for hosting the client and server sides' applications, respectively. Therefore, 16, 32 and 64 PMs are emulated during the above emulations. Two virtual machines (VMs) are configured on each PM with different roles (DataNode and NameNode) as a HDFS chunk server. The proposed and earlier versions of HDFS are installed on the mentioned virtual machines. To configure multiple virtual machines as HDFS chunk servers, the GNS3 VM remotely runs on a real server using VMware ESXi. The server is HP Proliant DL380 G9 [30]. The proposed and earlier versions of HDFS are installed on the mentioned virtual machines. Each VM in the cluster has 16 Intel Xeon CPU cores and 96 GB of DRAM running CentOS 7.5. Moreover, all VMs are connected to the emulated network via 10 GB Ethernet network interface. An 80 SSD GB storage is also used for data storage at each of the DataNodes. While 16, 32 and 64 subnets/clients have been emulated, the amount of overall generated data varies in the range of 160 GB and 3.2 TB. Since each DataNode contains an 80 SSD GB storage, TeraSort benchmark needs to access at least two DataNodes (under the management of coordination protocol) for accomplishing its jobs. Therefore, the overheads of using proposed coordination protocol have been imposed on different factors during the emulations. Local communication is handled by 2950 Cisco switches. Moreover, 2651 Cisco router is used as the subnet's gateway for remote communication. A Catalyst 3550 Cisco multilayer switch along with a route server acts as the Internet exchange point (IXP) for handling communication between subnets.

VMs can be configured as a conventional NameNode or DataNode. Also, they can be configured as a proposed chunk server consisting of both a NameNode slice and a DataNode. Hadoop 1.1.2 is selected as the base architecture for applying design decisions and ideas. The Intelligent Java IDE (IntelliJ IDEA) [17] is used for building modified java classes classified into Hadoop packages intended for implementations of file system for Hadoop over Web.

To measure the performance impact of the proposed architecture on the Hadoop infrastructure under real workloads, the standard TestDFSIO, TeraGen and TeraSort benchmarks [18] for different amounts of data are used. TestDFSIO benchmark is a read and write test for HDFS. It is helpful for tasks such as stress testing HDFS to discover performance bottlenecks in the network [18]. TeraGen and TeraSort benchmarks are used to test both MapReduce and HDFS by generating and sorting some amount of data to measure the capabilities of distributing and mapreducing files in cluster [18]. The compared HDFS Federation was developed in HDFS-1052 branch [9]. The new feature (federated NameNode architecture) has been merged into trunk and is available in 0.23 release [9]. Typically, some v-CPUs (virtual CPUs) should be allocated to GNS3-vms for configuring virtual machines. In the emulations, the GNS3 remotely runs on a real server using VMware ESXi. The server is HP Proliant DL380 G9. Each VM in the cluster has 16 Intel Xeon CPU cores and 96 GB of DRAM running CentOS 7.5. An increasing number of emulated subnets lead to reduced execution time of emulations due to limitation of infrastructure resources.

However, the comparison of proposed and earlier architecture of HDFS has been performed under the same conditions with respect to number of v-CPUs and infrastructure details. Therefore, the performance of GNS3 and limitation of infrastructure cannot affect the validation of comparisons.

6.2 Analysis and results

In this subsection, the results of using the proposed HDFS architecture (Elastic HDFS) are compared with the latest Apache HDFS distribution (HDFS Federation). The comparison is made with respect to two factors: performance and availability. Moreover, 10 GB, 30 GB and 50 GB data are fairly generated by client machines spread out over the emulated network.

6.2.1 Performance

In this paper, the primary metric used to measure performance of HDFS architecture is throughput. Throughput is the rate (bits per time unit) at which bits transferred between client machines and HDFS chunk servers. In order to compare the performance of Elastic HDFS and HDFS Federation in terms of throughput, experiments are performed under different congestion, failure and migration ratios.

Figure 11 shows the comparison of throughput of connections between client machines and HDFS chunk servers under the management of Elastic HDFS and HDFS Federation. The results are obtained with respect to different benchmarks, different numbers of subnets and different amounts of generated/requested data. Regarding the results of TestDFSIO read/write operations, read operation is a bit faster than write because of internal components of SSD disks. On the contrast of write throughput, the read throughput increases on average about 21.4% as the amount of data and number of subnets increase. As long as all the network bandwidth is allocated to the HDFS communication, the performance bottleneck can be just the processing delay of chunk servers. The requests received from the client machines to read some amount of data (blocks) impose less processing load on the chunk servers compared to write. The throughputs of TestDFSIO write, TeraGen and TeraSort downgrade about 24.0%, 26.2% and 8.4% while the amount of transferred data and number of subnets increase. All of the mentioned benchmarks involving write operation (TestDFSIO write, TeraGen and TeraSort) require more efforts from HDFS side.

Increasing amount of requested/generated data and number of subnets/clients lead to increase in network traffic. Under the above condition, the Elastic HDFS works better than HDFS Federation. The throughput of connections under the management of Elastic HDFS improves about 65.9%, 2.9% and 40.0% for TestDFSIO write, TeraGen and TeraSort, respectively. The coordination conducted by Elastic HDFS among chunks and its networking mechanism (anycast) are two reasons that can help shift the load from a congested chunk to the other ones. All of the chunk servers managed by Elastic HDFS can handle the requests via coordination messages

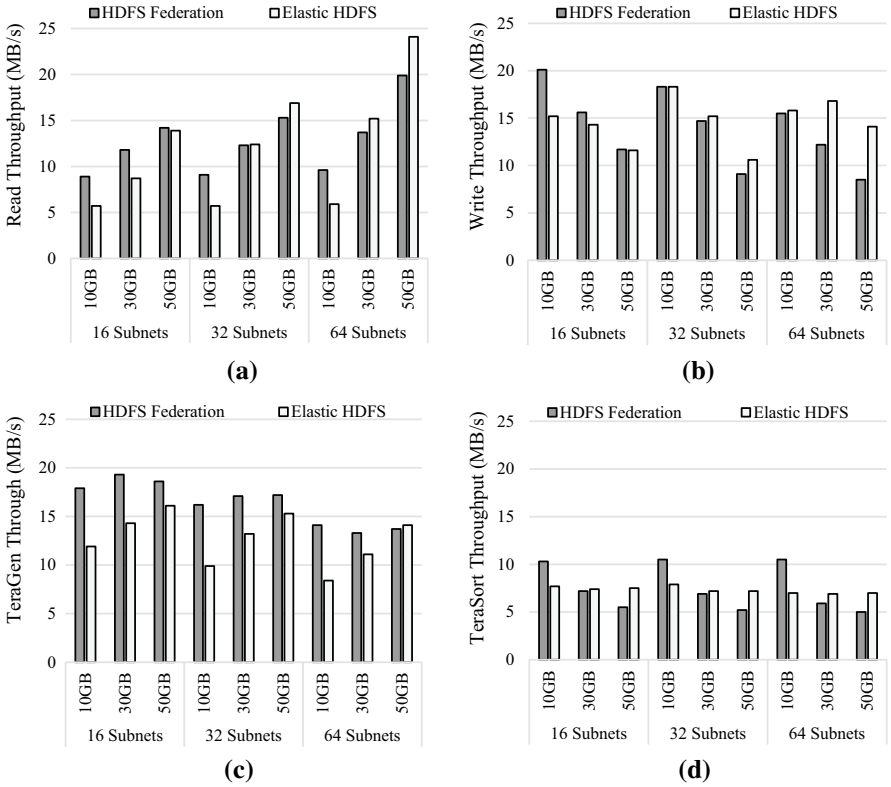

Fig. 11 Comparison of throughput of connections between client machines and HDFS chunk servers under the management of elastic HDFS and HDFS federation considering **a** TestDFSI read, **b** TestDFSIO write, **c** TeraGen and **d** TeraSort benchmarks

and protocol. The result is the improved traffic load distribution in the network and enhanced throughput.

With HDFS Federation, all of the requests should firstly pass through the NameNode clusters which are unfairly accessed by the nearer client machines. The cause is network/server overload and throughput degradation. However, the results indicate that the throughput of connections under the management of HDFS Federation is sometimes greater than the Elastic HDFS. For less amount of requested/generated data, the Federation architecture works better than Elastic HDFS. Compared to Elastic HDFS, the HDFS Federation is a no-frills architecture. While Elastic HDFS has been equipped with different mechanisms for tackling unstable network conditions, it imposes more additional traffic into network. In the case of stable network with low bandwidth usage without any problematic events, less complexity of Federation architecture has causes that the results are in the favor of Federation.

In the following, three separated analyses have been considered to compare the performance of Federation and Elastic architectures under different network conditions. The common point is that the Elastic architecture overcomes the Federation

while congestion, migration and failure ratio, as the signs of increased network instability, increase.

The above study was conducted to compare the throughput of the connections over a network without any secondary traffic. In order to examine the performance of the HDFS architectures on the network with different congestion conditions, a secondary traffic is generated and advertised over the network. Figure 12 shows the throughput of connections between client machines and chunk servers over the mentioned emulated network (16, 32 and 64 subnets) in terms of different congestion ratios. The TeraGen benchmark is selected for comparison. Considering the results of this benchmark for maximum number of subnets/clients (64) and maximum amount of generated data (50 GB), the throughputs of connections under the management of HDFS Federation and Elastic HDFS are closer to each other. Therefore, the effect of different levels of congestion on the performance of different HDFS architectures can be more clearly analyzed with the help of TeraGen. Regarding Fig. 12a–c, the throughput of connections between clients and chunk servers under the management of Elastic HDFS descends with lower slope in comparison with the HDFS Federation. Moreover, the crossover congestion ratio is about 53%, 36% and 32% on the network with 16, 32 and 64 subnets, respectively. As mentioned previously, the augmented number of subnets/clients leads to increased real traffic along with generated secondary traffic. Based on the results deduced, the throughput is less affected by the increased congestion ratio while Elastic HDFS is utilized. The coordination among different chunk servers conducted by Elastic HDFS can help distribute the traffic over the network in the favor of overloaded chunks. Concerning the defined functions and messages of coordination protocol, all of the chunks are able to redirect the requests to the chunks, which are responsible to handle them (Parents and Grandparents). However, the mentioned coordination is achieved through more processing efforts on the server side, causing less throughput compared to HDFS Federation on the un-congested networks.

In order to assess the robustness of different HDFS architectures against random failures, the throughput of connections managed by HDFS Federation and Elastic HDFS is investigated in the presence of different numbers of chunk server

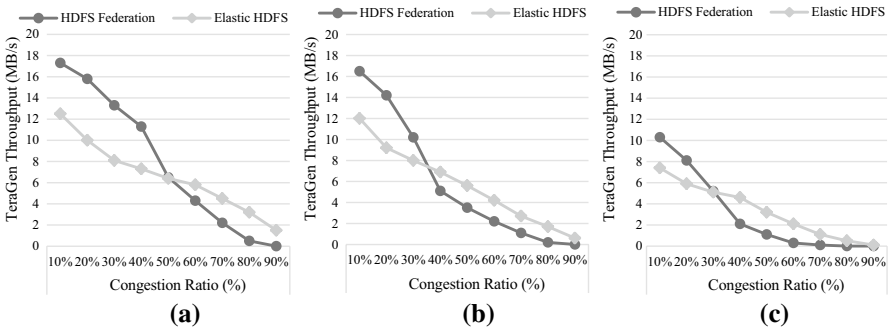

Fig. 12 Throughput of connections between client machines and chunk servers in terms of different congestion ratios over the mentioned emulated network with **a** 16 subnets, **b** 32 subnets and **c** 64 subnets

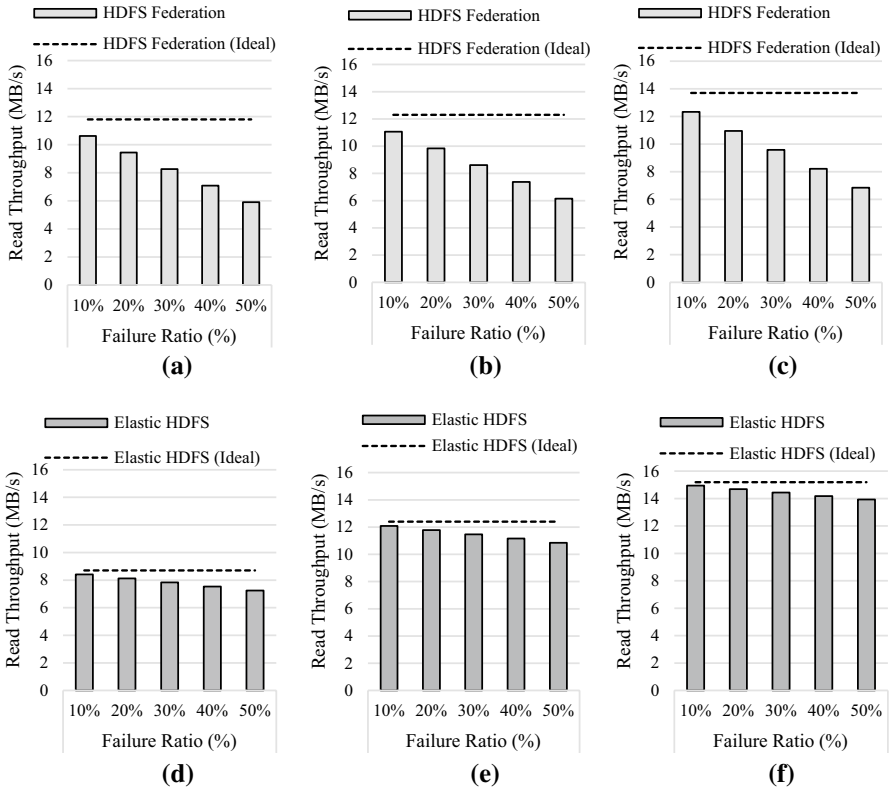

Fig. 13 Throughput of connections between client machines and chunk servers managed by HDFS federation and elastic HDFS in terms of different failure ratios of chunks for network with 16 subnets/clients (**a**, **d**), 32 subnets/clients (**b**, **e**) and 64 subnets/clients (**c**, **f**)

failures. Figure 13a–f compares the throughput of connections between client machines and chunk servers managed by HDFS Federation and Elastic HDFS in terms of different numbers of subnets/client and different failure ratios of chunks. Then, the results are compared with those extracted in the absence of failure. The ideal throughput line (dashed trend line in the charts) shows the throughput of connections between client machines and chunk servers without injecting any chunk server failure in the emulated network. The TestDFSIO read benchmark is selected for requesting 30 GB data from the chunk servers on the network with 16 subnets (Fig. 13a, d), 32 subnets (Fig. 13b, e) and 64 subnets (Fig. 13c, f). This purely read benchmark completely depends on the data and metadata that were previously generated and stored on the chunk servers. Consequently, TestDFSIO read is the one that is most vulnerable against chunk server failure among mentioned benchmarks. Regarding Fig. 13, the throughput of HDFS's connections managed by HDFS Federation almost linearly decreases when the number of chunk failures grows. On the other hand, Elastic HDFS shows more robustness against failures. As shown by Fig. 13b, d, the throughput degradation of

Elastic HDFS's connections is about 16.67%, 10.85% and 8.33% for 16, 32 and 64 subnets and worst-case scenario (50% failure ratio). The cooperation among chunk servers with different roles, implemented in the context of Elastic HDFS, improves the accessibility of data stored in the slaves in the presence of master node failure. Furthermore, the gap between the results of Elastic and Federation increases when the system scales up from 16 subnets/clients to 32 and 64 subnets/clients. In this case, the results of Elastic are also more closer to ideal values compared to the results of Federation. With the help of Elastic architecture, all of the chunk servers managed by Elastic HDFS can handle the requests via coordination messages and protocol, an ability that is not supported by Federation. Thus, the negative impacts of increasing failure rate on throughput of Elastic HDFS are smaller than its effects on HDFS Federation.

Finally, the throughput of HDFS connections under the management of Elastic HDFS and Federation architectures is investigated considering different client migration ratios. Figure 14a–c compares the throughput degradation of connections using Elastic HDFS and Federation architectures in terms of different number of client migrations. The results of TeraSort benchmark are selected for the above comparison in the case of using 16, 32 and 64 subnets/clients. TeraSort benchmark requires the highest number of client's accesses to metadata of stored data per each received request. Therefore, the effect of clients' movements over the emulated network on the throughput can be more clearly explainable via running this benchmark. The dashed trend lines in the charts represent the descending slope of throughput of connections managed by each architecture during migration ratio increment. In all scenarios, the throughput of Elastic HDFS's connections descends with lower slope in comparison with the results of Federation. The slope of throughput degradation of Federation's connections is about 1.72, 1.71 and 1.75 times greater than the throughput degradation of Elastic HDFS's connections. Therefore, it can be concluded that the Elastic HDFS can effectively handle a higher number of client migrations over the network.

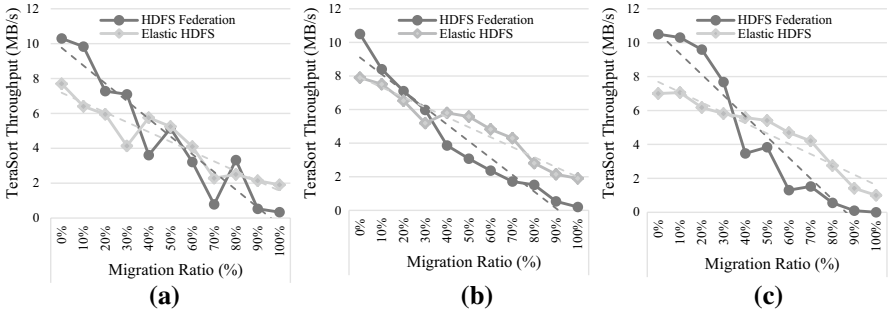

Fig. 14 Throughput degradation of connections using elastic HDFS and HDFS federation architectures in terms of different numbers of client migrations for network with **a** 16 subnets/clients, **b** 32 subnets/clients and **c** 64 subnets/clients

6.2.2 System availability

System availability is the probability a system is functioning when needed to, under normal operating conditions. When the system is alive and well structured, the organization can continue to produce output and meet orders. In other words, availability is the ratio of the system uptime to total functional time. From the client's viewpoint, the cloud storage should serve the request as quickly as a local storage. However, interactions between clients and cloud storage system suffer from some types of delay. The concentration of the following calculations is just on delay factors observed due to the architecture of cloud storage system. These delays increase Waiting Time of client for receiving its responses. Therefore, availability of a cloud storage system from the client's point of view can be calculated as:

$$\text{Availability} = \frac{\text{Total Time} - \text{Waiting Time}}{\text{Total Time}} \tag{1}$$

where Total Time is the total amount of time that a client is connected to the cloud storage system and Waiting Time is the total amount of time a client is waiting for receiving its responses. In other words, Waiting Time is the total amount of time intervals between sending requests and receiving corresponding responses. Waiting Time stems from three types of delays: Processing Delay, Network Delay and Connection Delay. It can be computed as:

$$\text{Waiting Time} = \text{Processing Delay} + \text{Network Delay} + \text{Connection Delay} \tag{2}$$

Processing Delay is the time required to commit a client's request by the cloud storage system. Type of the clients' requests and architecture of cloud storage system can increase/decrease the Processing Delay. Network Delay is the delay imposed by network core elements and depends on network conditions, such as queuing, network congestion, link failure and router failure. Finally, Connection Delay is the delay imposed because of the server connection conditions such as distances between client and chunk servers and congestion on the connections established for HDFS communications. While the architecture of a cloud storage system can improve and degrade the Connection and Processing Delays, it cannot affect the Network Delay.

In order to compare the effects of design decisions on the availability under different network conditions, three scenarios are taken into account with respect to different congestion, failure and migration ratios on the network with 64 subnets/clients. During the emulations, client machines select random addresses from namespace and send ten read requests for 1 GB data every 10 s over 1 min emulation process. TestDFSIO read benchmark is selected for availability assessment. This benchmark has the smallest size of client requests and needs less computing efforts by the YARN part of Hadoop. Subsequently, the effects of HDFS architectures on the Processing and Connection Delays can be more clearly analyzed via using this benchmark.

Figure 15a–c compares the availability of HDFS Federation and Elastic HDFS in terms of different congestion, migration and failure ratios. In the base condition (network without any secondary traffic and chunk failure), the Computing Delay imposed by the Elastic HDFS is about 12.3% more than the value imposed

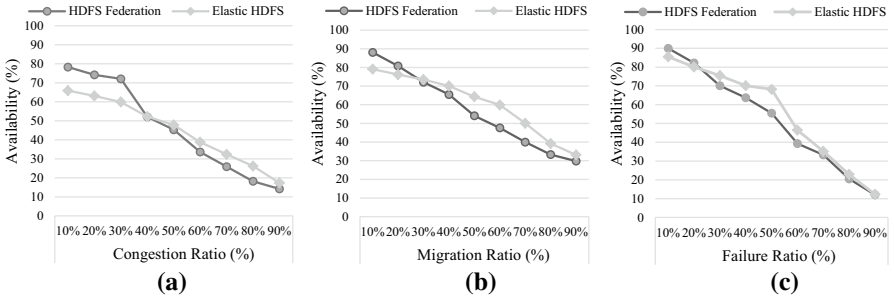

Fig. 15 Comparsion of the availability of HDFS Federation and elastic HDFS in terms of different congestion (**a**), migration (**b**) and failure (**c**) ratios

by HDFS Federation. This is due to delay of additional methods and messages designed and issued in the structure of Elastic HDFS for chunk servers' collaboration. Regarding Fig. 15a, the availability of Elastic HDFS is higher than the availability of Federation in the congested networks (for congestion ratio greater than 38.9%). In this case, the Connection Delay observed due to Federation is about 22.3% more than the Computing Delay imposed by Elastic HDFS. Under this condition, the Connection Delay imposed by the HDFS Federation increases significantly. This increment has greater negative impact on availability compared to the higher Processing Delay that incurs due to Elastic HDFS. While accesses to each namespace are just managed by one NameNode, congestion on its corresponding subnet leads to increase in number of blocked connections. Increasing the migration and failure ratios has similar impacts on Waiting Time. Regarding Fig. 15a, b, the migration ratio of 28.5% and the failure ratio of 23.2% are the crossover points from which the availability of Elastic HDFS dominates the availability of HDFS Federation. Coordination among HDFS chunks conducted by Elastic HDFS architecture causes that other chunks could deal with client requests in the absence of responsible ones due to failure or client migration. Although the coordination among HDFS chunks established by Elastic HDFS increases the processing efforts on the server side, it can help reduce the dependency of overall server-side operations on the specified NameNode. The maximum availability improvements of the proposed architecture during the mentioned scenarios are about 43.9%, 25.8% and 22.9% for network with different congestion, migration and failure ratios, respectively. These improvements are observed at about congestion ratio of 79.3%, migration ratio of 58.2% and failure ratio of 50.0%.

Finally, it should be noted that the size of G-fsimage considered for storing owner tags in Grandparent is, on average, about 33.34% of F-fsimage during emulations. G-fsimage contains addresses of Parent just for the resided objects that are children of other chunks. While size of overall namespace directories in emulations is about 17% of whole data capacity of chunk servers, the storage overhead of Elastic HDFS is, on average, about 5.67% of whole data capacity. Therefore, the aforementioned availability and throughput enhancements are achieved with negligible storage overhead.

7 Conclusion

This paper focuses on the scalability limitation of commercial off-the-shelf (COTS) storage systems. The main concentration is on enhancement of conventional architecture to remove the performance bottleneck and single point of failure problem. To accomplish the goal, a scalable distributed architecture is presented for Hadoop distributed file system (HDFS) as a common cloud storage system to reduce dependency of overall cloud-side operations on the single master node. The centralized structure of namespace directory is changed to interconnected distributed model via splitting namespace directory into slices and distributing over chunk servers. Each chunk server is responsible for handling some clients' requests related to its corresponding namespace slice. Chunk servers communicate with each other to commit request via a coordination protocol. With the help of proposed architecture, all of the chunk servers can handle the requests via coordination messages and protocol, an ability that is not supported by previous version of HDFS. In the case of chunk server failure and unstable network conditions, coordination of chunk servers helps clients keep accesses to slaves without significant disruption. The result is the improved traffic load distribution in the network and enhanced throughput. The experimental results demonstrate that the proposed architecture causes that the descending slopes of throughput decrease by 32.6%, 77.4% and 42.1% during growth of congestion ratio, failure ratio and migration ratio. The maximum availability improvements of the proposed architecture are about 43.9%, 25.8% and 22.9% for network with different congestion, migration and failure ratios, respectively. Future studies could fruitfully explore this issue further by assessment of the effects of distributed architecture on different factors related to cloud storage systems such as usability, cost and security. Moreover, future research on the overhead of coordination protocol might extend the explanation of some drawbacks of the interconnected distributed architecture.

Acknowledgements The present study was supported by Golestan University (Grant 981871), Gorgan, Iran.

References

1. Cai H et al (2016) IoT-based big data storage systems in cloud computing: perspectives and challenges. IEEE Internet Things J 4(1):75–78
2. Mahmood T et al (2018) Karma: cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency. IEEE Trans Cloud Comput 1(1):18–28
3. Mittal A et al (2015) Google file system and Hadoop distributed file system: an analogy. Int J Innov Adv Comput Sci 4(1):29–43
4. Hu D et al (2015) Research on reliability of Hadoop distributed file system. Int J Multimed Ubiquitous Eng 10(11):42–54
5. Iliadis I et al (2014) Reliability of geo-replicated cloud storage systems. In: 2014 IEEE Pacific Rim International Symposium on Dependable Computing, Singapore, pp 169–179
6. Asif Khan M et al (2012) Highly available Hadoop namenode architecture. In: 2012 International Conference on Advanced Computer Science Applications and Technologies, Malaysia, pp 167–172

7. Liu J et al (2016) Reliable and confidential cloud storage with efficient data forwarding functionality. IET Commun J 10(6):661–668
8. Xing L et al (2017) Reliability modeling of mesh storage area networks for Internet of Things. IEEE Internet Things J 4(6):2047–2057
9. HDFS Federation (2018) Retrieved 1 Mar 2019 from https://hadoop.apache.org/docs/r2.7.7/hadoop-project-dist/hadoop-hdfs/Federation.html
10. Uber (2018) Retrieved 1 Mar 2019 from https://www.uber.com/
11. Hakimzadeh K et al (2014) Scaling HDFS with a strongly consistent relational model for metadata. In: 2014 IFIP International Conference on Distributed Applications and Interoperable Systems, Germany, pp 19–31
12. Huang Z (2014) DNN: a distributed namenode filesystem for Hadoop. In partial fulfilment of requirements for the degree of Master of Science, University of Nebraska–Lincoln
13. Kim Y et al (2014) A distributed namenode cluster for a highly-available Hadoop distributed file system. In: 2014 IEEE International Symposium on Reliable Distributed Systems, Japan, pp 835–851
14. Xue R et al (2014) Partitioner: a distributed HDFS metadata server cluster. In: 2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, China, pp 167–174
15. Wang Y et al (2012) Clover: a distributed file system of expandable metadata service derived from HDFS. In: 2012 IEEE International Conference on Cluster Computing, USA, pp 126–134
16. Graphical Network Simulator 3 (2018) Retrieved 20 Sep 2018 from https://www.gns3.com/
17. Intelligent Java IDE (2016) Retrieved 15 Feb 2019 from https://www.jetbrains.com/idea/
18. R. Nayak (2018) Hadoop performance evaluation by benchmarking and stress testing with TeraSort and TestDFSIO, Retrieved 1 Mar 2019 from https://medium.com/ymedialabs-innovation/hadoop-performance-evaluation-by-benchmarking-and-stress-testing-with-terasort-and-testdfsio-444b22c77db2
19. Apache Hadoop version 1.x.y (2012) Retrieved 2 Mar 2019 from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
20. Apache Hadoop version 2.x.y (2015) Retrieved 2 Mar 2019 from https://hadoop.apache.org/docs/r2.7.2/
21. HDFS High Availability Using the Quorum Journal Manager (2017) Retrieved 1 Mar 2019 from https://hadoop.apache.org/docs/r2.7.7/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html
22. Apache Hadoop version 3.x.y (2017) Retrieved 1 Mar 2019 from https://hadoop.apache.org/docs/r3.0.0/
23. Gupta T et al (2015) An extended HDFS with an AVATAR NODE to handle both small files and to eliminate single point of Failure. In: 2015 International Conference on Soft Computing Techniques and Implementations, India, pp 67–71
24. Wang Z et al (2013) NCluster: using multiple active namenodes to achieve high availability for HDFS. In: 2013 IEEE International Conference on High Performance Computing and Communications, China, pp 2291–2297
25. Tang Y et al (2015) MICS: mingling chained storage combining replication and erasure coding. In: 2015 IEEE Symposium on Reliable Distributed Systems, Canada, pp 192–201
26. Yin J et al (2017) ASSER: an efficient, reliable, and cost-effective storage scheme for object-based cloud storage systems. IEEE Trans Comput 66(8):1326–1340
27. Application Request Routing (2018) Retrieved 10 Dec 2018 from https://www.iis.net/downloads/microsoft/application-request-routing
28. NGINX (2018) Retrieved 10 Dec 2018 from https://nginx.org/en/
29. Wang C et al (2013) Privacy-preserving public auditing for secure cloud storage. IEEE Trans Comput 62(2):362–375
30. HPE ProLiant DL380 Gen9 Server (2017) Retrieved 20 Feb 2019 from https://www.hpe.com/us/en/product-catalog/servers/proliant-servers/pip.hpe-proliant-dl380-gen9-server.7271241.html