2023 IEEE 41st International Conference on Computer Design (ICCD) | 979-8-3503-4291-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCD58817.2023.00051

Low-Latency and Scalable Full-path Indexing Metadata Service for Distributed File Systems

Chao Dong, Fang Wang*, Yuxin Yang, Mengya Lei, Jianshun Zhang, Dan Feng
WNLO, Key Laboratory of Information Storage System, Ministry of Education of China
Huazhong University of Science & Technology, Wuhan, China
Wangfang@mail.hust.edu.cn, {chaosdong, yuxinyang, lmy_up, shunzi, dfeng}@hust.edu.cn

***Abstract*—Distributed file systems (DFS) are the cornerstone of modern mass data processing systems. In DFS, the metadata service, as the core component, often becomes a performance bottleneck. Existing metadata service solutions have implemented flattened metadata management and full-path indexing to achieve high scalability in terms of capacity and throughput. However, these approaches have limitations, such as conflicts with POSIX-style permission verification and flawed support for super directories, leading to high and unstable latency that cannot provide reliable service for latency-sensitive applications. To overcome these limitations, we propose *Duplex*, a scalable DFS metadata service based on full-path indexing, which aims for low and stable latency. *Duplex* incorporates three key designs: a fast access path featuring a centralized permission server for efficient permission verification, a permission merging algorithm to reduce the PMS's space footprint, and flattened metadata management based on double consistent hashing that enables low-latency access to super directories. Our evaluations demonstrate that, compared to state-of-the-art metadata solutions, *Duplex* significantly reduces the average lookup latency by up to 84% and the 99th percentile tail latency by up to 88.2% for metadata-intensive benchmarks. Additionally, *Duplex* improves the lookup IOPS by up to** $7.6\times/2.3\times$ **compared to CephFS and BeeGFS.**

***Index Terms*—Distributed file system, Metadata Service, Latency, Full-path Indexing**

I. Introduction

Due to high throughput, high scalability and high availability, the large-scale distributed file system (DFS) has been widely deployed in modern data centers to serve data-intensive applications, e.g., big data analytics [4], high-performance computing (HPC) [5], cloud computing [7], [11], AI training, and web applications. These scenarios can be classified into two categories, throughput-sensitive applications and latency-sensitive applications. Throughput-sensitive applications, such as checkpointing for HPC and batch access for AI training, require storage backends with high I/O throughput and scalable capacity. Conversely, latency-sensitive applications, such as financial storage and interactive VR/AR/MR, demand low and stable operation latency to meet stringent service-level objectives (SLOs).

In DFS, metadata access is on the critical data path and accounts for more than half of all system accesses [3]. Consequently, the metadata service is always a performance bottleneck [13], [4], which is particularly problematic for large-scale DFS with petabyte-scale metadata volumes in modern data centers [3]. To achieve high scalability in terms of capacity and throughput, current metadata solutions tend to use full-path indexing based on flattened metadata management. Flattened metadata management uses hash algorithms and key-value databases to manage metadata in a unified flattened namespace, building a flexible and scalable storage architecture for metadata [9], [12]. The full-path indexing method is based on flattened metadata management. It uses the full path of files as indexes, shortening access path and increasing throughput [20]. However, despite their success in achieving scalable capacity and throughput, these techniques face two challenges in meeting latency service-level objectives (SLOs) for latency-sensitive applications:

This work was supported by NSFC (No. U22A2027, 61832020, 61821003) and project 2023BAA018.

- *Conflict between permission checking and full-path indexing.* In POSIX-compatible file systems, each component has its own permissions. Once a file is requested, the file system checks permissions using a component-based method, which splits the pathname into components (separated names of directories and files) and traverses them sequentially in the hierarchical directory tree [10]. However, in large-scale DFS, the directory tree is distributed across multiple metadata servers (MDS), requiring the metadata traversal to go through multiple hops among MDSs, resulting in high latency for remote access. The hierarchical permission mechanism negates the performance gain from the full-path indexing method. To shorten latency, state-of-the-art schemes use parallel path resolution to access permissions of all components on the same path in parallel [12], [13]. However, this approach maintains many network connections with varying delays when accessing deep files, still suffering from long tail latency.
- *Unpredictable latency in super directories.* The flattened metadata management prototypes treat each file as a separate object and distribute them across multiple MDS, resulting in high load balance but damaging spatial locality. Unfortunately, spatial locality is crucial for file system performance, particularly for range operations performed at the directory level such as `ls` and `rmdir`. To balance spatial locality and load balance, current flattened metadata management schemes distribute metadata at the directory level or directory subset level, instead of in single-file granularity [16], [12], [15]. However, these approaches result in access hotspots and long tail latency for super directories that contain a large number of files.

To address the issues incurring high and unpredictable access latency, this paper presents *Duplex*, an innovative metadata service for large-scale DFS. *Duplex* aims to deliver low and stable manipulation latency for latency-sensitive applications while maintaining scalable throughput and capacity for throughput-sensitive applications. The solution includes three key designs to optimize the indexing mechanism and metadata management. First, *Duplex* adopts a novel metadata service architecture with a dual access path approach, featuring a fast path with low latency and a slow path with high throughput. Both paths support complete POSIX-style permission verification. The fast path includes a dedicated permission server (PMS) that caches all directory permissions from the MDSs, enabling requests from latency-sensitive applications to complete permission checks quickly. Second, *Duplex* proposes a tree-based permission merging algorithm to improve the space efficiency of the single-node PMS, preventing the PMS from being the system capacity bottleneck. Third, *Duplex* uses a novel flattened metadata management scheme that distributes directory subsets among the MDS clusters by double consistent hashing (DCH), providing low and stable access latency even for deep files and super directories.

In summary, this paper makes the following contributions:

- Identification of the limitations present in state-of-the-art metadata service schemes when aiming to achieve stringent latency SLOs (§II).
- Introduction of a novel metadata service solution, named *Duplex*, featuring an innovative architecture comprising a fast path and a slow path catering to latency-sensitive and throughput-sensitive applications, respectively (§III-A). On the fast path, *Duplex* incorporates a dedicated yet non-scalable PMS equipped with a permission merging algorithm to facilitate rapid permission checks (§III-B). On the slow path, the MDS cluster adopts a novel flattened metadata management scheme that leverages parallel path resolution to achieve scalable throughput (§III-C).
- Implementation of *Duplex* in a distributed cluster based on the IndexFS source [16] and comprehensive performance evaluations comparing it with state-of-the-art metadata service designs (§IV).

II. Background and Motivation

Large-scale distributed file systems (DFS) are designed to efficiently store and manage vast volumes of data across numerous networked nodes [14], [19]. These nodes collaborate to provide a unified file system view for multiple clients. In the context of DFS, metadata refers to essential information that describes the structure and location of files and directories within the system, encompassing details like names, attributes, ownership, and physical location. Extensive research has demonstrated that metadata operations are highly prevalent within DFS, with statistics showing that 50% to 80% of all file system accesses involve metadata-related actions [1], [3]. Consequently, the efficient provision and management of high-quality metadata services are of paramount importance for optimizing the overall performance of DFS.

A. Traditional hierarchical metadata service

Traditional file systems organize metadata using a hierarchical tree-based structure. The directory tree comprises directories that serve as containers for both files and subdirectories. In this structure, branch nodes are directories, while leaf nodes represent files. GoogleFS [4], for instance, manages the entire directory tree within a dedicated MDS. However, this centralized approach can lead to performance bottlenecks and scalability challenges. To enhance scalability, some DFSs distribute the metadata service across multiple servers at the subtree level [13], [19]. The directory tree is divided into multiple subtrees, with each subtree assigned to a distinct server for management. However, this method results in imbalanced workloads among MDSs, particularly when workloads dynamically change [17].

Hierarchical metadata services rely on the component-based lookup method. When a client needs to access file metadata, it divides the file path into multiple components (separated by ”/”) and sequentially traverses the directory tree layer by layer, starting from the root directory and progressing toward the target file [10]. While this approach is effective in local file systems, it poses challenges in DFS. Traversing a path involves multiple MDSs and incurs substantial remote overhead, contributing to increased file access times [16], [9]. Moreover, the component-based lookup method can lead to access hotspots in DFSs due to the need to access the root directory for every operation.

To address the limitations associated with hierarchical metadata management and component-based lookup method, flattened metadata management and full-path indexing method have been proposed.

B. Flattened metadata management

In contrast to tree-based distribution, a flattened metadata service distributes metadata across different MDSs using random algorithms like hashing. Typically, each metadata server manages its local metadata in a single table or database. Flattened metadata management offers increased flexibility and efficiency in metadata operations, particularly in large-scale systems. File system operations, including file lookup, creation, and deletion, can be performed more efficiently due to easier access and modification of metadata. Furthermore, this approach simplifies metadata distribution across multiple servers, resulting in enhanced scalability and fault tolerance.

LocoFS [9] represents a typical flattened metadata service for DFS, which transforms directory tree metadata into separate objects stored in key-value databases. This transformation leads to reduced latency and increased throughput. However, LocoFS distributes metadata at the file granularity, sacrificing the spacial locality of directories. Range operations at the directory level, such as `ls` and `rmdir`, rely on a single-node directory metadata server, limiting scalability. InfiniFS [12] improving range operation performance by distributing metadata at the directory granularity, ensuring that files within the same directory are on the same MDS. However, this approach does not adequately consider the challenges posed by super directories. In large-scale file systems, super directories, which contain millions of files, can create significant activity hotspots when assigned to a single MDS. To mitigate this, Patil et al. introduced *GIGA+* [15], a system that partitions super directories into subsets and distributes them across different MDSs. IndexFS [16] adopted this approach to achieve high scalability and balanced workloads. However, *GIGA+*'s client often communicates with multiple MDSs to obtain the desired mapping for locating files within a super directory, resulting in increased latency. Hence, current flattened metadata management schemes struggle to meet stringent latency SLOs when accessing super directories.

C. Full-path indexing method

The full-path indexing method, facilitated by flattened metadata management, stands in contrast to the traditional component-based lookup method, which relies on parent-child relationships between directories. In the full-path indexing method, all files and directories reside within the same flattened namespace, allowing for rapid lookup and retrieval of files based on their complete path [20], [8]. While the full-path indexing method holds promise in reducing lookup latency, it encounters challenges in file systems due to conflicts with the POSIX-style permission mechanism. In systems adhering to POSIX standards, permissions are managed through a component-based method, where each directory or file possesses its individual permissions. The file system determines the final permission for a file by traversing the components along the file path and sequentially checking their permissions. Given the critical role of permission checks in maintaining security, the permission query still necessitates traversal through hierarchical components, negating the performance advantages offered by the full-path indexing method. Consequently, reconciling conflicts between full-path indexing and POSIX-style permissions presents a formidable challenge [2], [18].

Several research efforts have been dedicated to addressing this challenge. One approach involves storing the complete set of permissions with each file. For example, CalvinFS [18] associates all ancestor directory permissions with each file to support hierarchical access control in the POSIX standard. While this approach eliminates the need to traverse the entire directory tree to check distributed permissions, it introduces significant storage overhead due to redundant permission metadata in large-scale DFS deployments. Besides, modifying directory permissions, particularly for directories closer to the root, entails recursively propagating the new permissions to all descendants of the modified directory. Another solution is parallel path resolution, as employed by InfiniFS [12] and HopsFS [13]. These systems use parallel path resolution to access the permissions of components along the file path simultaneously. Consequently, a file access operation can be completed within a single round-trip time (RTT). However, when accessing deeply nested files, the client likely maintains hundreds of remote connections for one request, incurring huge overhead on CPU and network. Furthermore, network conditions in practice may not be consistent across different nodes, and the latency of parallel path resolution depends on the longest RTT. Regrettably, these full-path indexing methods are unable to guarantee strict latency SLOs.

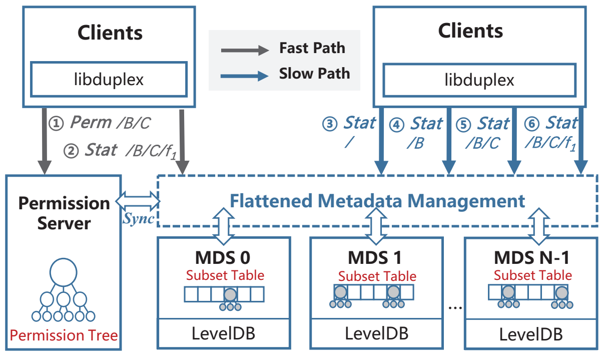

Fig. 1. Architecture of Duplex

III. Design

This paper introduces *Duplex*, a scalable filesystem metadata service with full-path indexing, featuring three key design elements. To address the conflicts between full-path indexing and POSIX-style permission mechanisms, *Duplex* employs a novel architecture that offers a fast access path for latency-sensitive applications and a slower path for throughput-sensitive applications. The fast path integrates a dedicated Permission Server (PMS) that caches directory permissions to expedite permission queries (§III-A). Additionally, *Duplex* incorporates a tree-based permission merging algorithm for the PMS to reduce its space footprint (§III-B). Lastly, it utilizes a scalable flattened metadata management system that prioritizes both balance and locality, ensuring rapid access to super directories (§III-C).

A. Architecture

Figure 1 illustrates the architecture of *Duplex*, consisting of three primary components:

Metadata Servers (MDSs) play a pivotal role in *Duplex*, responsible for distributed metadata services. The directory tree is managed across MDSs through a flattened directory-subset-grained distribution scheme. To optimize performance, metadata resides in the memory of MDSs, with metadata updates recorded as logs persisting in local levelDB storage for reliability.

Permission Server (PMS): *Duplex* decouples directory data into contents (*timestamp*, *size*, and *child list*) and permissions (*user*, *group*, and *mode*). During initialization, directory permissions are copied from the MDS cluster to a dedicated permission server (PMS). Hierarchical directory permissions are centrally managed by the PMS to facilitate permission checks. By providing the full path of a designated file, the PMS verifies the permissions of the file's ancestor directories and returns the result. *Duplex* employs a write-through mode to maintain consistency between the MDS cluster and the PMS. Any update to directory permissions is first applied to the MDS cluster and then synchronized with the PMS, ensuring crash consistency. In case of the PMS crash and restart, hierarchical permissions can be reconstructed from the MDS cluster.

Client: Clients of *Duplex* execute file system operations through the userspace client library *libduplex*. Clients support two access methods: a fast path, which offers low latency but limited scalability, and a relatively slower but more scalable path.

- Fast path. In the fast path, the PMS handles permission checks, and the target file is retrieved from the MDS cluster based on the full path. Figure 1 illustrates an example where a client accesses `/B/C/`f_1 using the fast path. Initially, the PMS verifies the permissions of the path `/B/C` (①), while the MDS cluster verifies f_1's permissions (②). Only after passing both checks can the client access the desired file metadata from the MDS cluster. The two permission checks can be processed in parallel, further reducing the access delay.
- Slow path. The slow path in *Duplex* caters to throughput-sensitive applications that can tolerate higher latency. This path allows clients to bypass the PMS and access the MDS cluster directly through parallel path resolution. For instance, when accessing `/B/C/`f_1, clients on the slow path access multiple components in parallel, including `/` (③), `/B` (④), `/B/C` (⑤), and `/B/C/`f_1 (⑥). Only after successfully passing all permission checks can the client retrieve the metadata of `/B/C/`f_1 from the MDS cluster.

B. *Tree-based permission management*

The design of the PMS effectively mitigates permission overhead. However, the single-node PMS storing permission metadata for the entire system has scalability limitations. To address this challenge, *Duplex* employs two methods to enhance the space efficiency of the PMS and enable it to support large-scale DFS deployments: storing only directory permissions (§III-B1) and merging identical permissions (§III-B2). As the organization of permissions changes, the processing flow of permission operations also adapts (§III-B3).

1) *Storing directory permissions only*

The PMS exclusively manages directory permissions since clients can retrieve file metadata, including file permissions, from MDSs using full-path indexing. Given that directory metadata accounts for less than $\frac{1}{10}$ of the entire file system metadata [3], and permissions represent only a fraction of directory metadata, directory permissions constitute a mere 2% of the total metadata volume in *Duplex*.

Figure 2(b) illustrates how the PMS extracts directory permissions from the directory tree. The directory tree depicted in Figure 2(a) comprises 9 directories with four types of permissions and 5 files. The PMS eliminates unnecessary metadata, such as file metadata and directory contents, retaining only directory permissions, as shown in Figure 2(b). This representation is referred to as the permission tree in this paper.

2) *Merging identical permissions.*

Duplex introduces a permission merging algorithm to further reduce storage requirements. In file systems, subdirectories typically inherit the same permissions as their parent directory. This characteristic is leveraged to eliminate redundant permissions from the permission tree. The process from Figure 2(b) to Figure 2(c) illustrates the consolidation of a permission tree. Initially, entries with permissions identical to their parent are

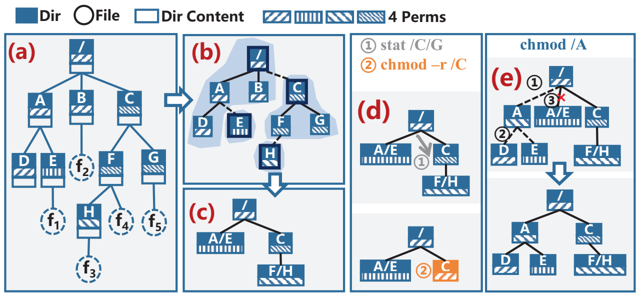

Fig. 2. (a) The original directory tree; (b) The initial permission tree generated from *(a)*; (c) The merged permission tree from *(b)*; (d) A lookup and a recursive update to *(c)*; (e) A non-recursive update to *(c)*.

pruned from the tree, resulting in the retention of 4 directories: `/`, `/A/E`, `/C`, and `/C/F/H`. Subsequently, path prefixes that match parent paths are omitted. The node indexes transform into `/`, `A/E`, `C`, and `F/H`. The resulting permission tree adopts the form of a radix tree (or compact prefix tree), which is an efficient data structure for storing and retrieving strings [6].

3) *Processing permission operations.*

This section illustrates how the permission tree handles permission operations, emphasizing that these operations pertain exclusively to directories since the PMS manages directory permissions exclusively.

A. Lookup. Permission lookup stands as one of the fundamental and critical functions, as it should be performed before any metadata request to ensure safety. The PMS conducts lookups using a top-down prefix matching process, resembling a radix tree. If a node ID matches the prefix components of the requested pathname, the PMS enters the node to perform further prefix matching. The lookup process concludes when no node matches the remaining pathname. Figure 2(d) provides an example of querying the permission for directory `/C/G`. The PMS successively accesses nodes `/` and `C`. However, none of the children of `C` matches the remaining path `G`. Consequently, the PMS determines that `G` shares the same permission with `C`, and thus, it checks the permissions of `/` and `C`.

B. Insert. Analogous to the *lookup* operation, the PMS processes *insert* operations through a top-down prefix matching process. When the search reaches the nodes that match the last components of the inserted pathname, three possible scenarios arise. First, if the node matches the inserted pathname exactly, an error is raised because the directory to be added already exists. Second, if the permissions of the current node are identical to the inserted permissions, no updates are made, as the new permission duplicates that of its parent. Third, if the permissions differ, a new node is created as a child of the current node. The new node's ID corresponds to the remaining directory pathname, and its value represents the new permission. For example, if a client inserts directory `/A/E/I` with distinct permissions from `/A/E` in Figure 2(c), a new node `I` will be generated as the child of `/A/E`.

C. Remove. Removing a directory necessitates the recursive removal of its children. For instance, the removal of directory `/C` in Figure 2(c) involves removing nodes `/C` and `F/H`.

Similarly, the removal of directory `/C/F` in Figure 2 entails removing node `F/H`.

D. Update. Permission updates can be either *recursive* or *non-recursive*. Recursive updates entail modifying the permissions of all files and subdirectories beneath the given directory to the same value. To accomplish this, the PMS consolidates the involved directories into a single node. Therefore, a *recursive update* operation is essentially a combination of a *Remove* operation followed by an *Insert* operation. For instance, operation ② in Figure 2(d) recursively alters the permissions of directory `/C` by initially removing it and subsequently inserting it with the new permissions.

In contrast, non-recursive updates are more intricate, particularly when the updated directory has previously been merged into the ancestor but must now be re-added as an individual node. Figure 2(e) illustrates a non-recursive update to directory `/A` in Figure 2(c). Firstly, the PMS creates a new node if no node matches the directory's pathname. In the absence of a match, a node `A` is established as the child of node `/` (①). Secondly, the PMS retrieves the permissions of the subdirectories of the designated directory from the MDS cluster and inserts them into the permission tree. As a result, nodes `D` and `E` are added as children of node `A` (②). Lastly, the PMS adjusts the permission tree using the permission merging algorithm, which includes removing nodes with identical permissions as their parents and reorganizing the neighboring nodes of the manipulated node. In Figure 2(e), node `A/E` is removed since it is redundant to `E` (③). The final permission tree is presented at the bottom of Figure 2(e).

The merged permission tree enhances storage efficiency and ensures the accuracy of permission operations. Moreover, the permission merging algorithm reduces the depth of the tree, resulting in fewer hierarchical searches and improved permission performance.

C. Flattened metadata management

Duplex employs a novel flattened metadata management approach to efficiently handle large-scale DFS metadata within the MDS cluster. The distribution of the directory tree is accomplished through three steps. Firstly, the directory tree is partitioned into directories. Secondly, each directory is divided into *subset*s, and file metadata is dispersed across the subsets using consistent hashing (CH) (§III-C1). Finally, the subsets are mapped to MDSs based on a second CH calculation (§III-C2). Similarly, clients utilize double CH calculations to index files from the MDS cluster, which is termed as DCH in this paper (§III-C3).

1) Partitioning directories into subsets.

Some metadata services partition the directory tree at the directory level, leading to activity hotspots in super directories [12], [5]. To address this issue, *Duplex* divides directories into finer-grained units called directory subsets.

Each directory is organized into one or more subsets. Figure 3(a) illustrates an example of partitioning a directory tree with 6 directories and 5 files into 7 subsets. Directory `A` is divided into two subsets, while each of the other directories is managed as a single subset. Determining the number of subsets for a directory involves considering two key parameters: *directory size* and *maximum subset size*. *Directory size* refers to the total number of files, including sub-directories, within a directory, denoted as *DirSize*. For instance, in Figure 3(a), directory `A` contains 3 files: f_1, `C`, and `D`, resulting in a *DirSize* of 3. The *maximum subset size*, denoted as S_{max}, represents an upper limit on the number of files a subset can contain. The number of subsets required to partition a directory is calculated as $\boldsymbol{n} = \lfloor DirSize/S_{max} \rfloor + 1$. In the example of Figure 3(a), with S_{max} set to 3, directory `A` is divided into two subsets because its *DirSize* is 3, reaching the limit. Other directories with *DirSize* less than 3 are each managed as a single subset. A subset comprises three components:

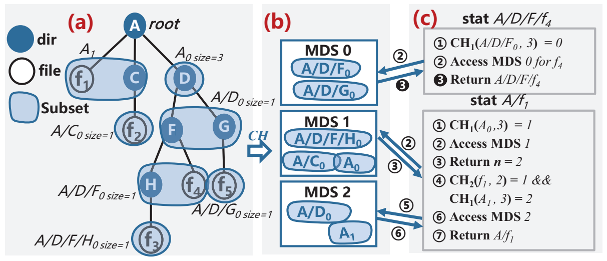

Fig. 3. (a) Partitioning a directory tree into 7 subsets ($S_{max} = 3$); (b) Distributing 7 subsets among three MDSs; (c) Accessing files from *(b)* through DCH.

- *ID*. Each subset is identified by an ID consisting of two parts: the pathname of the corresponding directory to distinguish subsets from different directories and a unique number that increments from 0 to distinguish subsets within the same directory. In Figure 3(a), the two subsets of `A` are identified as A_0 and A_1, respectively.
- *Size*. The directory size is centrally stored in the *master subset* (*mSubset*), which is the first subset (subset `0`) of the directory. Each directory has exactly one *mSubset*. In Figure 3(a), the size of directory `A` is stored in subset A_0.
- *Files*. When a directory is divided into multiple subsets, files within the directory are evenly distributed across the subsets based on the CH results of their file names. As shown in Figure 3(a), directory `A` has two subsets that contain metadata for its children. For example, `D` is located in subset A_0 because CH(`D`, 2)=0, while f_1 and `C` are in subset A_1 because their CH values are `1`. The use of CH ensures minimal metadata migration when adjusting subsets due to changes in directory size.

2) Distributing subsets among MDSs.

After dividing directories into subsets, *Duplex* distributes these subsets among MDSs using a second CH algorithm. By providing subset IDs as input, the subsets are assigned to the respective MDSs. This secondary CH ensures load balance and minimizes subset migration when adjustments are made to MDS assignments. Figure 3(b) provides an example of distributing seven subsets among three MDSs.

3) DCH for file indexing.

The *mSubset* functions as the fixed access entry point for its corresponding directory. Figure 3(c) illustrates how files, such as `A/D/F/`f_4 and `A/`f_1, are indexed in the MDS cluster. Let's first consider the example of `A/D/F/`f_4. The client initiates access to the *mSubset* by default. The subset ID is determined by combining the directory path with the subset ID `0`, resulting in `A/D/`F_0. The client then employs a CH algorithm to calculate the target MDS ID, which yields `0` (①). Subsequently, the client accesses MDS `0` (②). Since `A/D/F` contains only one subset, the requested file, `A/D/`F_0, is successfully retrieved (❸), completing the lookup process.

Next, let's consider the example of `stat A/`f_1. The first two steps remain the same, where the client communicates with the *mSubset* A_0 on MDS `1` for the file `A/`f_1 (① and ②). However, as the file is not in the subset, MDS `1` returns the number of subsets instead of the desired file metadata. Since $S_{max} = 3$ and the size of directory `A` is `3`, which is recorded in subset A_0, MDS calculates that directory `A` contains two subsets and returns $n = 2$ to the client (③). At this point, the client employs the DCH method, which involves two CH calculations to locate files. The first calculation (CH_1) determines the subset ID based on the file name, revealing that file `A/`f_1 is in subset `1`. The second calculation is for the MDS ID based on the subset ID. The client inputs A_1 into CH_2 and determines that the target MDS ID is `2` (④). Finally, the client retrieves file `A/`f_1 from MDS `2` (⑤ and ⑥).

The DCH approach ensures that clients can access a file within two RTTs, regardless of the file depth and directory size.

IV. Evaluation

In this section, we evaluate the performance of *Duplex*. The evaluation is divided into two parts based on different evaluation methods. In the first part, we use MDTest as a benchmark to measure metadata operation performance, including throughput, latency, and load balance (§IV-B). In the second part, we compare different permission mechanisms using static file system metadata collected from private production servers (§IV-C).

A. Experimental Setting

Duplex is based on the IndexFS source code and implemented in *C++*, comprising approximately 4k lines of modified code. IndexFS is a distributed metadata service for large-scale file systems, which uses *GIGA+* [15] (a hash-based management) to partition the directory tree and a component-based lookup method to index files. We have entirely altered metadata distribution and indexing methods while reusing the LevelDB-based journal mechanism from IndexFS. All experiments are performed on identical commercial machines, each equipped with 24 processors and 32 GB of memory. As shown in Table I, the nodes run CentOS 7.6 and are connected by 40-GB InfiniBand networks with a round-trip time of approximately 120 us. For optimal performance, we employ ram-disks as the storage devices.

TABLE I
ENVIRONMENT CONFIGURATION

Processor	Intel Xeon CPU E5-2620 0 @ 2.10GHz * 24
Memory	32 GB
Storage	RamDisk
OS	CentOS Linux release 7.6.1810
Kernel	3.10.0-957.el7.x86_64
Compiler	GCC 4.8.5 20150623
Network	ConnectX Dual Port 40Gb/s InfiniBand

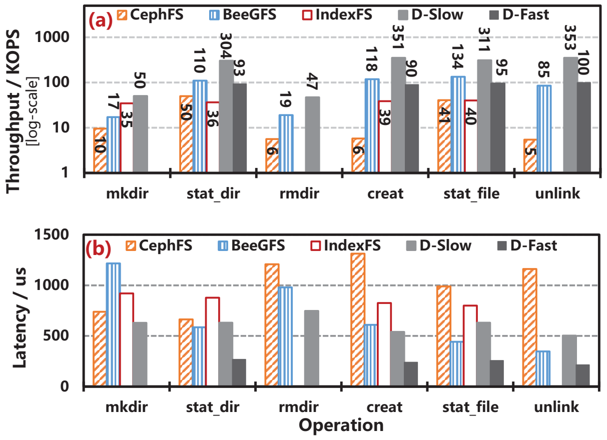

Fig. 4. Comparison of throughput (a) and latency (b) for CephFS, BeeGFS, IndexFS, and Duplex (D-Slow and D-Fast) within a 16-node cluster for six types of operations.

B. MdTest Benchmark

Comparison Schemes: In this section, we compare the performance of four metadata services. The first scheme is a 16-node CephFS, consisting of 16 OSDs and 16 MDSs. All MDSs are active, resulting in a distributed directory tree across the 16 nodes. The second metadata service scheme is BeeGFS [5] version 7.0, one of the leading parallel file systems. BeeGFS consists of 1 manager process, 16 MDS processes, and 16 storage processes on 16 physical nodes. The third scheme is IndexFS [16], comprising 16 physical nodes, each running an MDS process. All three schemes use the component-based lookup method. The fourth scheme is our *Duplex* with 16 MDSs and 1 PMS. The MDSs are deployed on 16 physical nodes, and the PMS is on MDS `0`. *Duplex* is tested both within the slow path (*D-Slow*) and fast path (*D-Fast*).

Benchmark: We use the MDTest benchmark tool, which generates a highly concurrent stream of metadata operations on a file system. We initialize MDTest client processes in parallel across 16 nodes via MPI. Each MDTest process generates 100K requests across 512 directories with a depth of 9 in the directory tree. We test six types of operations, including *mkdir*, *stat_dir*, *rmdir*, *creat*, *stat_file*, and *unlink*. The first three are directory operations, and the rest are for files. Since IndexFS's open-source code has not implemented *rmdir* and *unlink*, we exclude these two operations from the IndexFS evaluation. Besides, the difference between *D-Fast* and *D-Slow* is the permission checking method, and they share the same process of *mkdir* and *rmdir*. Hence, we exclude these two operations from *D-Fast* evaluation. We compare the performance in terms of throughput, load balance, and latency.

1) Throughput.

We conduct throughput tests by scaling the number of MDTest processes from 1 to 64 per node to measure the maximum throughput provided by each scheme, with a maximum of $16 \times 64 = 1024$ MDTest clients. The throughput results are displayed in Figure 4(a), with the x-axis denoting the six types of operations, and the y-axis representing the corresponding throughput of different schemes, where KOPS stands for 1000 operations per second.

We can make three observations from the figure:a) *Duplex* significantly outperforms other schemes in terms of throughput for file operations. The throughput of *D-Slow* is about 350 KOPS, which is $2.3\times-4.1\times$ higher than that of BeeGFS. The throughput performance benefits from the highly scalable flattened metadata management. *Duplex*'s clients can directly access target MDSs without cross-server coordination. b) For `mkdir` and `rmdir`, the throughput of *Duplex* is only about 50 KOPS. This is because `mkdir` and `rmdir` update directory permissions, requiring synchronization with the single-node PMS, which becomes a throughput bottleneck for these operations. Although the throughput is relatively low, it is still over $2.3\times$ higher than that of CephFS and BeeGFS. c) The lookup throughput of *D-Fast* is also limited by PMS's performance. However, its throughput is higher than that of `mkdir` and `rmdir`, since lookups occur with less concurrency overhead to the PMS than updates.

2) *Latency.*

In this section, we present a comparison of the latency of the four metadata service solutions. The experimental setup is the same as in the previous section, except that we run only one MDTest client process on each node to avoid congestion. Figure 4(b) shows the average latency of the four schemes. The x-axis denotes the six types of operations, and the y-axis is the corresponding latency. The results show that *Duplex* with the fast path consistently has the lowest average latencies for all six operations when compared to the other schemes. In particular, the `stat` latency of *D-Fast* is only 200 us, leading to a reduction of up to 74.1% compared to CephFS and 41.9% compared to BeeGFS.

3) *Load Balancing.*

In this section, we compare the load balancing capability of the 4 schemes. We collect the number of inodes on each of the 16 MDSs during the last experiment as MDSs' loads. Then, we normalize the loads to the average value. The load distribution is shown in Figure 5, where the x-axis represents the ID of the 16 MDSs, denoted by `1` to `f` in hexadecimal. The y-axis shows the normalized loads on each MDS, where their average value is 1. From the figure, we observe that the loads of CephFS and BeeGFS are heavily skewed, as they use tree-based partitioning with coarse partition granularity. Differently, IndexFS and Duplex achieve almost full load balance because they use hash-based partitioning.

C. *Static file system traces*

Benchmark: To evaluate our new permission mechanism, we collected three static file system traces from our private production servers. Each trace was collected by traversing the directory tree from the root directory and accessing all directories and files in the operating system. The trace includes the names and metadata of the directories and files but excludes the file content as it does not affect the test results. We collected three traces under different workloads: SQL databases and web services in CentOS, data backup in Fedora, and desktop applications in Windows. Each trace contains around 100,000 directories and 0.37-1 million files with depths of 6-9.

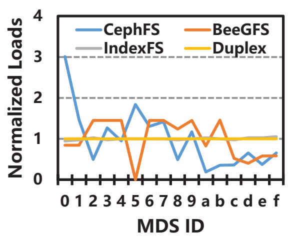

Fig. 5. Load balance comparison of CephFS, BeeGFS, IndexFS, and Duplex within a 16-node cluster

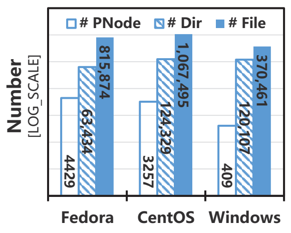

Fig. 6. Scale of the 3 traces and the generated permission trees

Comparison Schemes: There are three comparison schemes. The first scheme is IndexFS, which is still under testing with 16 nodes. IndexFS uses a component-based lookup method to check permissions on the path sequentially, and this scheme is denoted by *Index*. The second scheme is the slow path of *Duplex*, which is denoted by *D-Slow*. The third scheme is the fast path of *Duplex* and is denoted by *D-Fast*.

1) *Permission Merging Algorithm.*

We loaded the three traces into *Duplex* and generated permission trees on the PMS. To demonstrate the compression ratio of the permission merging algorithm, we compared the size of the permission tree on the PMS with that of the original directory tree. The results are presented in Figure 6, where we indicate the number of files (# File) and directories (# Dir) of each trace and record the total number of nodes in the merged permission tree (# PNode). Since the numbers vary widely, we measured the y-axis on a log scale.

From Figure 6, we can see that the PNode scale is significantly smaller than the size of the original directory tree. For example, in the CentOS trace, the number of PNodes is 3,276, while the number of directories is roughly 123 thousand, and the number of files is about 1 million. The number of PNodes accounts for only 2.6% of the number of directories and 0.3% of the number of files. These results confirm the efficiency of the permission merging algorithm. In theory, a PMS can support hundreds of MDSs.

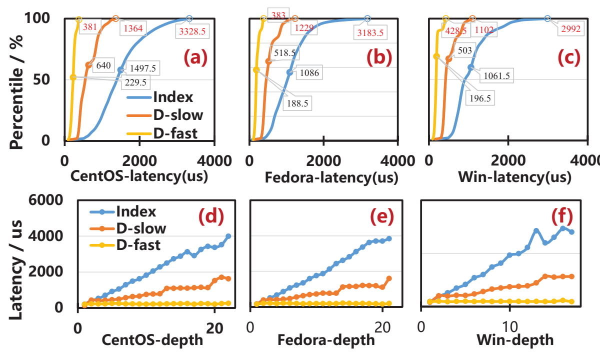

Fig. 7. Latency comparison of IndexFS, the slow path of *Duplex*, and the fast path of *Duplex* within a 16-node cluster after loading static file system traces. (a-c) CDF of `stat` latency. (d-f) `Stat` latency for different file depths.

2) Latency Comparison.

To demonstrate the superiority of *Duplex*, we evaluate the latency of *stat* operations with complete permission verification. We follow the steps outlined below for each data trace: First, we load the data trace into the three metadata service schemes. Then, the client randomly selects 50,000 files from the data trace and accesses their metadata using the `stat` commands. We record the latency of each `stat` operation to show the access overhead, including the time for permission checks. We then sort the 50,000 `stat` records and illustrate the cumulative distribution function (CDF) of the latency results in Figures 7(a-c). These figures also show the average latency and 99th percentile (P99) latency. Furthermore, we classify records by accessed depth and illustrate the operational latency at different file depths in Figures 7(d-f).

The results shown in Figures 7(a-c) demonstrate that the fast path of *Duplex* outperforms the other two comparison schemes in terms of both average latency and P99 latency. For instance, in the *CentOS* trace, the average latency of *D-Fast* is 229.5 us, which is 84% lower than IndexFS and 64.2% lower than *D-Slow*. The P99 latency of *D-Fast* is 392 us, which is 88.2% lower than Index and 71.3% lower than *D-Slow*.

Figures 7(d-f) illustrate the average access latency at different file depths. We observe that the access latency of *Index* and *D-Slow* increases linearly as the file depth increases. For example, in the CentOS trace, as the file depths increase from 1 to 20, the latency of *Index* increases from 212 us to 3000+ us, and the latency of *D-Slow* increases from 202 us to 1625 us. IndexFS uses the sequential component-based lookup method, which involves too many remote communications during permission checking, leading to poor performance. *D-Slow* uses a parallel access method to reduce latency, resulting in moderate performance. In contrast, *D-Fast* uses a dedicated PMS to serve permission checking and locate files via full-path indexing, leading to low and stable access latency.

V. Conclusions

Duplex is a scalable metadata service for distributed file systems that aims at low and stable metadata access latency. Through its novel designs, such as the single-node PMS and full-path indexing based on DCH, Duplex has demonstrated low latency and high scalability in evaluations. Duplex represents a promising approach to metadata management for distributed file systems. Future work could build on its design to further improve efficiency and performance.

Acknowledgements

The author would like to express sincere gratitude to Professor Hong Jiang from the University of Texas at Arlington for his invaluable contributions to improving the writing of this paper. This work was supported by National Natural Science Foundation of China (NSFC) No. U22A2027, 61832020, 61821003, and project 2023BAA018.

References

[1] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3):9–es, 2007.

[2] Miao Cai, Junru Shen, Bin Tang, Hao Huang, and Baoliu Ye. FlatFS: Flatten hierarchical file system namespace on non-volatile memories. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 899–914, Carlsbad, CA, July 2022. USENIX Association.

[3] Hao Dai, Yang Wang, Kenneth B. Kent, et al. The state of the art of metadata managements in large-scale distributed file systems — scalability, performance and availability. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3850–3869, 2022.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[5] Jan Heichler. An introduction to beegfs, 2014.

[6] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[7] Qiang Li, Qiao Xiang, Yuxin Wang, et al. More than capacity: Performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, Santa Clara, CA, February 2023. USENIX Association.

[8] Siyang Li, Fenlin Liu, Jiwu Shu, Youyou Lu, Tao Li, and Yang Hu. A flattened metadata service for distributed file systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2641–2657, 2018.

[9] Siyang Li, Youyou Lu, Jiwu Shu, et al. Locofs: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[10] Linux. Filesystems in the linux kernel - pathname lookup, 2022. https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html Accessed July 11, 2022.

[11] Haifeng Liu, Wei Ding, Yuan Chen, et al. Cfs: A distributed file system for large scale container platforms. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1729–1742, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Wenhao Lv, Youyou Lu, Yiming Zhang, et al. InfiniFS: An efficient metadata service for Large-Scale distributed filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.

[13] Salman Niazi, Mahmoud Ismail, Seif Haridi, et al. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, February 2017. USENIX Association.

[14] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[15] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, February 2011. USENIX Association.

[16] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.

[17] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. Mantle: a programmable metadata load balancer for the ceph file system. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[18] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, Santa Clara, CA, February 2015. USENIX Association.

[19] Sage A Weil. *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, University of California, Santa Cruz, 2007.

[20] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E Porter, and Jun Yuan. The full path to full-path indexing. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 123–138, 2018.