Journal of Universal Computer Science, vol. 26, no. 1 (2020), 89-106
submitted: 30/12/18, accepted: 15/11/19, appeared: 28/1/20

Scalable Distributed Metadata Server Based on Nonblocking Transactions

Kohei Hiraga
(Keio University, Fujisawa, Kanagawa, Japan
hiraga@keio.jp)

Osamu Tatebe
(University of Tsukuba, Tsukuba, Ibaraki, Japan
tatebe@cs.tsukuba.ac.jp)

Hideyuki Kawashima
(Keio University, Fujisawa, Kanagawa, Japan
river@sfc.keio.ac.jp)

Abstract: Metadata performance scalability is critically important in high-performance computing when accessing many small files from millions of clients. This paper proposes a design of a scalable distributed metadata server, *PPMDS*, for parallel file systems using multiple key-value servers. In PPMDS, hierarchical namespace of a file system is efficiently managed by multiple servers. Multiple entries can be atomically updated using a nonblocking distributed transaction based on an algorithm of dynamic software transactional memory. This paper also proposes optimizations to further improve the metadata performance by introducing a server-side transaction processing, multiple readers, and a shared lock mode, which reduce the number of remote procedure calls and prevent unnecessary blocking. Performance evaluation shows the scalable performance up to 3 servers, and achieves 62,000 operations per second, which is 2.58x performance improvement compared to a single metadata performance.

Key Words: distributed metadata server, nonblocking distributed transaction, parallel file system
Category: D.4.2, D.4.3, D.4.8

1 Introduction

In high-performance computing (HPC), the computational performance has been improved by increasing the number of CPU cores and compute nodes. The storage performance also needs to be improved in sync, otherwise, the performance gap between CPU and storage becomes wider and wider. To fill the gap, there are two problems; I/O bandwidth and metadata performance. The I/O bandwidth, in principle, can be solved by using arrays of high-performance storages such as NVMe SSD and persistent memory, while the metadata performance cannot be solved easily. The metadata is internal data that manages parallel file systems such as hierarchical namespace, file or block locations, timestamps, and access control information. Improving the metadata performance means, for example, to improve file creation performance, how many files can be created in a second. To improve the metadata performance, the most difficult issue is how to efficiently manage the hierarchical namespace in parallel.

One of the solutions of this problem is data partitioning. Subtree partitioning [Weil et al. 2006] that partitions hierarchical namespace by subtrees among multiple servers, improves the scalability of a metadata server in parallel file systems. It can improve metadata performance when accessing different subtrees, while it does not improve the performance when accessing the same subtree. HPC applications often create and open millions of file in a single directory. In this case, the subtree partitioning does not help to improve the metadata performance.

Hash partitioning [Schmuck and Haskin 2002, Patil and Gibson 2011] partitions entries in a single directory among multiple servers using a hash function. This helps to improve the metadata performance for creating millions of files in a single directory since it just creates an entry in the corresponding server, while a certain type of metadata operations such as moving a file and removing a directory, requires consistency in data (i.e., the status of files) among multiple servers. Clearly it requires distributed consistency protocol. Two-phase commit [Gray 1978] is widely used for this purpose, and it performs efficiently if blocking phenomenon does not occur. However, the transaction process will block when one of the participants fails or cannot communicate with the coordinator node due to, for example, network partitioning. This shortcoming has been addressed in literature [Samaras et al. 1993].

This paper proposes PPMDS, a scalable distributed metadata server for a parallel file system, which improves metadata performance and scalability without sacrificing file system semantics using the nonblocking transaction scheme. It expresses the logical tree structure as key-value pairs to efficiently manage hierarchical namespace in a file system across multiple servers. The key field is a set of a parent inode number and an entry name. The remaining value field contains a file metadata. In this design, entries can be partitioned using a hash function, which enables parallel file creation, parallel file removal, and parallel file references.

To update multiple inode entries atomically and efficiently, PPMDS exploits a nonblocking distributed transaction processing scheme based on the dynamic software transactional memory [Herlihy et al. 2003] without using problematic two-phase commit. This enables directory removal and directory listing consistently. PPMDS further improves the metadata performance by introducing the server-side transaction processing, the open-for-read scheme for multiple readers, and the shared lock mode, which reduces the number of remote procedure calls and avoids unnecessary blocking time. Performance evaluation shows the scalable performance up to 3 servers, and achieves 62,000 operations per second when creating files in a single directory, which is 2.58x performance improvement compared to a single metadata performance.

The contributions of this paper are follows;

- Design and implementation of scalable metadata servers without sacrificing file system consistency,
- Application of nonblocking transactions for key-value stores,
- Enhancement to improve transaction processing, and
- Prototype implementation that shows the scalable performance in file operations in a single directory.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 describes the design of the PPMDS. Section 4 describes the nonblocking transaction, which is a key issue for the design of the PPMDS. Section 5 describes the implementation of the PPMDS. Section 6 shows the evaluation result. Section 7 concludes this paper.

2 Related Work

Distributed file systems utilize a dedicated metadata server approach for high efficiency. Examples are Lustre [Braam], Gfarm [Tatebe et al. 2010], Google File System [Ghemawat et al. 2003], and HDFS [Hadoop]. These file systems have a master metadata server that stores all file system metadata in memory. Since all metadata is in memory, these systems provide efficient metadata performance but has a limitation in terms of memory capacity and scalability. The number of entries to be managed is limited by the memory capacity, and the performance is limited by a single node performance.

To improve the metadata performance and scalability, subtree partitioning [Weil et al. 2006] and hash partitioning [Schmuck and Haskin 2002] of the hierarchical namespace are used. The subtree partitioning is efficient when each client accesses different subtrees. On the other hand, the hash partitioning can help to improve the metadata performance in a single directory. However, there is a challenging problem to implement file system operations, such as removing a directory and moving a directory, that require to modify multiple metadata across multiple servers atomically.

Two-phase commit has been widely used for this purpose. It requires additional message exchanges that include vote request, vote commit or abort, and global commit or abort among a coordinator and all participants. Xiong et al. reduced the overhead by combining the two-phase commit algorithm with metadata processing [Xiong et al. 2011]. Two-phase commit has drawbacks not only for the performance but also for the recovery process. To recover from a node failure, a coordinator and all participants need transaction logs, and all participants require a log of global committed transactions from the coordinator.

Our proposed method does not use the two-phase commit since it has several drawbacks, but uses a nonblocking distributed transaction based on an algorithm of dynamic software transactional memory. The nonblocking feature prevents from blocking the whole transaction process even when some node fails.

Ursa Minor [Sinnamohideen et al. 2010] makes a choice not to use the distributed transaction. It migrates related metadata into a single metadata server, while it does not solve the scalable issue.

There are several file systems that make a choice not to use the distributed transaction such as IndexFS [Ren et al. 2014], BatchFS [Zheng et al. 2014] and DeltaFS [Zheng et al. 2015]. Instead, they limit the functionality of the file system. That is different from our method that does not restrict the file system functionality by using the distributed transactions.

HopsFS [Niazi et al. 2017] utilizes shared-nothing, transactional, in-memory NewSQL databases for metadata servers. It supports distributed transactions among metadata servers, but it optimizes the metadata read operations specially for the Spotify music streaming workloads. Our proposed method works efficiently not only the metadata read operations but also write operations.

PPFS [Takatsu et al. 2017] is a distributed file system that utilizes the PPMDS [Hiraga et al. 2018] for file system metadata management. It introduces the design of the PPFS file system using the PPMDS and PPOST object storage-based file servers [Takatsu et al. 2016]. It includes the performance evaluation of not only the PPFS but also the PPMDS comparing with the IndexFS. According to the paper, the PPMDS shows 2.5x better performance and better scalability than the IndexFS for directory creation workload using 5 servers and 128 clients.

This paper extends the earlier design and implementation of the PPMDS metadata server [Hiraga et al. 2018] that exploits nonblocking distributed transactions.

3 Design of PPMDS Distributed Metadata Server

Our design goal is to achieve scalable performance for file operations by multiple metadata servers without restricting the file system functionality. For this goal, the challenging issue is how to manage hierarchical namespace efficiently in parallel.

Traditional local file system uses directory entries to manage a hierarchical name space, which is a list (or a tree) of an entry name and the inode number. This data structure is not appropriate for distributed servers. In PPMDS, there is no directory entry that manages entries in a directory. Instead, entries in a directory are distributed across multiple servers. Only we need to manage is a list of servers where entries in a directory are stored. Another challenging issue is how to support all file system functionalities. For example, a directory removal operation removes a directory only when there is no entry under the directory. This operation should be atomic, that is, there is no other operation between the file existence check and the directory removal. If it is not atomic, the inconsistency happens. When a file creation happens between the file existence check and the directory removal, the newly created file will not have a parent directory. To avoid this situation, PPMDS uses nonblocking distributed transaction among multiple servers.

This design can eliminate unnecessary blocking when operations are not conflicted. File creation and removal for different files are not conflicted. Directory operations such as directory creation and removal, that modify multiple entries atomically can be supported by the nonblocking transaction efficiently. This design eliminates a global lock or a distributed lock in a directory level, and ensures to process concurrent operations in parallel unless they are conflicting operations.

3.1 Lookup Operation and Data Structure

Traditional file systems are optimized for a block device in a single machine, which are managed by the inode data structure. However, since it includes many indirections, it is not efficient in distributed environment.

A file system metadata consists of two types of data; file metadata such as a list of block locations, permission and timestamps, and hierarchical namespace.

In the traditional file system, the hierarchical namespace is managed by directory entries, which has several problems in distributed environment and also parallel updates. When a directory entry is managed in a single server, it will be a bottleneck when many updates happen in a single directory simultaneously. In PPMDS, there is no directory entry. The file system metadata is managed by key-value pairs. Each key-value pair is basically an inode entry. The file metadata is stored as a value of a key-value pair. Key is a pair of a parent inode number and an entry name. This design makes directory listing possible by range-based query among all servers without explicit directory entries. Inode entry is identified by a pair of a parent inode number and an entry name.

Figure 1 shows an example to look up *"/dir-x/file-a"*.

1. Look up the root inode entry by specifying *(inode=0, "/")* as a key. Since the root directory is a special case such that there is no parent directory, *inode=0* is specified.

2. Check the permission of the root directory for looking up the entry. If it is permitted, obtain an inode entry of the root directory that includes the

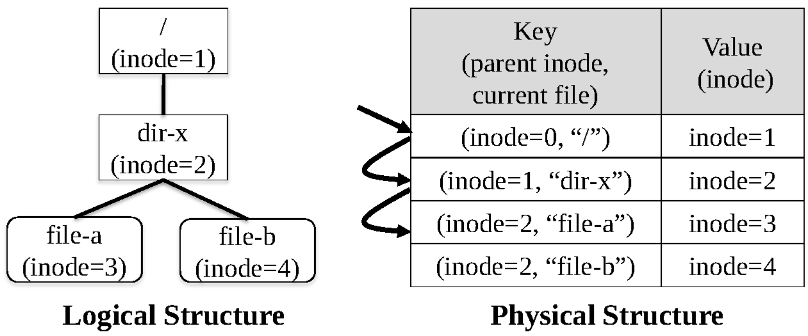

Figure 1: Lookup Operation in PPMDS

inode number *inode=1*.

3. Look up an inode entry of *“/dir-x”* by specifying *(inode=1, “dir-x”)* as a key. Check the permission, and obtain an inode entry that includes the inode number *inode=2* if it is permitted.

4. Look up an inode entry of *“/dir-x/file-a”* by specifying *(inode=2, “file-a”)* as a key. Check the permission, and obtain an inode entry of *“/dir-x/file-a”*.

For directory listing of the *“/dir-x”* directory, the range-based query is used as follows;

1. Look up the *“/dir-x”* directory, and obtain the inode number *inode=2*.

2. Obtain inode entries in the *“/dir-x”* directory by range-based query by *(inode=2, *)*.

3.2 Distributed Data Structure of Metadata

PPMDS manages the metadata using key-value pairs, which can be naturally distributed over multiple metadata servers. Each metadata server has its own key-value store.

In this case, when listing a directory, the range-based query is required for all servers that store entries in the directory. When the directory has many entries, it makes sense to store them in many servers, however, when the directory has small number of entries, requesting the range-query for many servers causes an overhead to retrieve small number of entries. This means the number of servers to be distributed depends on the number of entries in a directory. That is why we manage a list of distributed servers for each directory.

To manage a list of distributed servers of a directory, it is stored in all servers besides the data structure of metadata described in Section 3.1. It is created for each directory, and stored in the same key-value store of the data structure of metadata. Key is an inode number of the directory, and the value is a list of distributed server identifiers that store inode entries in the directory. Inode entries in the directory are distributed using a hash function among metadata servers in the list. The reason why all metadata servers have a server list is we would like to avoid the traffic concentration when looking up an entry. If there is one server list, all lookup processes access the single server list, which causes a performance bottleneck. To remove the bottleneck when looking up entries, we decide to store it in all servers. This requires additional overhead to create a directory, but we prioritize the lookup performance rather than directory creation performance since directory creations are rare in HPC applications.

3.3 Transaction of File Creation

To create a new file, a transactional scheme is necessary to keep consistency of the metadata in PPMDS. A transaction should be isolated from other transactions. During a transaction, some operation may fail unexpectedly due to, for example, the node failure. Even in this case, a distributed system is expected to continue the service. However, if the system does not respond to the expectation, uncommitted operations should not be reflected to the metadata database. To satisfy the requirements, PPMDS provides distributed transaction processing.

The following is an example to create a new file *"/dir-x/file-c"*;

1. A client looks up the parent directory *"/dir-x"*, and obtains *inode=2*. Then, the client sends a file creation request of *(inode=2, "file-c")* to a PPMDS server.

2. The requested server (α) starts a transaction for the file creation request. α obtains a list of servers from the local key-value store requesting *inode=2* as a key.

3. α determines a target server (β) for creating *"file-c"* among the list of servers. α requests β to create an entry of *(inode=2, "file-c")*. β creates the entry and responds to α. Finally α commits the transaction.

The transaction enables to modify several key-value pairs stored in multiple servers atomically. Details for the transaction will be described in Section 4. In step 1, a client can send a request to any PPMDS server, on the other hand, the number of remote procedure calls in the transaction can be reduced when it sends to the PPMDS server where the requested entry will be stored.

A transaction for a file creation obtains a list of distributed servers in step 2, and creates a metadata entry in step 3. These operations are not conflicted when multiple clients request to create different entries in the same directory. Multiple clients can create different entries in the same directory simultaneously with the nonblocking transaction processing scheme described in Section 4.

3.4 Transaction of Directory Creation

Creating a new directory also requires a transaction. The following is an example to create a new directory *"/dir-y"*;

1. A client looks up the root directory *"/"*, and obtains *inode=1*. Then, the client sends a directory creation request for *(inode=1, "dir-y")* to a PPMDS server.
2. The selected PPMDS server (α) starts a transaction for the directory creation. α obtains a list of distributed servers of the directory from the local key-value store requesting *inode=1* as a key.
3. α determines a target server (β) for creating *"dir-y"* among the list of servers. α requests β to create an entry of *(inode=1, "dir-y")*.
4. β determines a list of distributed servers for the new directory.
5. β generates a unique inode number, and creates a new entry for the directory in all the related servers, where the key is a unique inode number and the value is a list of distributed servers. β responds to α.
6. α commits the transaction.

Creating a directory requires an additional step 5 to create a list of distributed servers in all the related servers. This is a design choice. Directory operations are much rarer than file operations in HPC applications. In step 4, a server list is determined using currently available servers that can be dynamically managed by a reliable distributed coordination system such as Chubby [Burrows 2006] and ZooKeeper [Hunt et al. 2010]. In step 5, a new inode number should be unique in the system. A unique key consists of a server identifier and a serial number in each server that can be generated by each PPMDS server independently.

4 Nonblocking Transaction in PPMDS

Herlihy et al. proposed the nonblocking transaction scheme that is an efficient distributed transaction processing technique based on dynamic software transactional memory [Herlihy et al. 2003]. Kumazaki et al. extended the nonblocking transaction scheme using a key-value store [Kumazaki et al. 2011]. We utilize

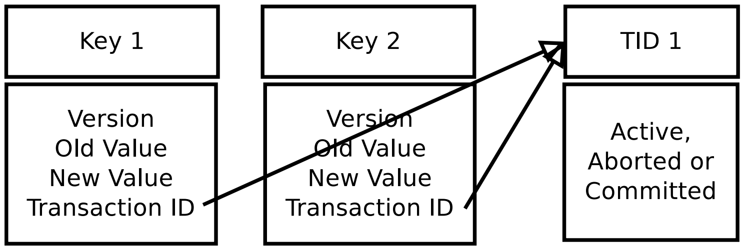

Figure 2: Data Structure of Transactional Key-value Pair

this nonblocking transaction scheme for a key-value store, and improve the performance with several extensions including server-side transaction processing, multiple readers, and shared lock mode.

Herlihy's nonblocking transaction provides serializable isolation level. This means transactions that include a series of operations, behave to be executed in a serial order even though each server processes transactions concurrently. Herlihy's nonblocking transaction is obstruction free. This means it does not block when it progresses alone. The obstruction freedom is nonblocking as well as lock freedom and wait freedom. A nonblocking transaction has a property such that a failed or delayed transaction does not prevent progress of other transactions.

To support the nonblocking transaction, a key-value pair has a data structure depicted in Figure 2, which is called a *transactional key-value pair*. Each key-value pair has a key, a version, an old value, a new value and a *Transaction ID*. The version means the number of writes to the transactional key-value pair, which is our extension to support multiple readers and shared lock mode described in Subsections 4.2 and 4.3. Transaction ID shows an *Owner* of the key-value pair. A transaction is managed by its identifier (transaction ID) and its status information. The transaction ID is also managed in a key-value store as depicted in Figure 2. There are three states for a transaction; *Committed*, *Aborted*, and *Active*. If the state is *committed*, the transaction is already committed. Then, *new value* is the up-to-date value. If the state is *aborted*, the transaction was aborted. Then, *old value* is the up-to-date value. Finally, if the state is *active*, the transaction is in progress.

To begin a transaction, a transactional key-value pair should be *open*ed to get the ownership. When the transaction state is committed or aborted, the status is changed to active using a compare-and-swap atomic operation. When it can be changed to active, the open succeeds, and the version is incremented. When the compare-and-swap operation fails or the transaction status is already active, the open fails. This means another transaction is currently in progress. In this case, there are two choices, to wait for the termination of the transaction, or to steal the ownership. To steal the ownership, the status is changed to aborted using the compare-and-swap atomic operation. To change the status, the compare-and-swap atomic operation is always used to avoid the race condition. For this decision, any conflict resolution algorithm is possible if it satisfies the obstruction freedom. One of algorithms that satisfy is a backing off algorithm. It waits for a while, changing the wait time double, and finally it steals the ownership when it exceeds the maximum wait time.

To commit the transaction, the status is changed from active to committed using the compare-and-swap atomic operation. If this compare-and-swap fails, the transaction has been aborted by another transaction.

4.1 Server-side Transaction Processing

When a client starts a transaction, there are many remote procedure calls to servers for each operation to access key-value pairs. Moreover, at the beginning of the transaction, and at the end, the client needs to change the state of the transaction stored in a remote server by the compare-and-swap atomic operation.

However, this series of remote procedure calls can be reduced when it is executed at the server side. Accessing to key-value pairs can be processed locally in a server. In this case, a client just asks to start a transaction to a server. For example, to create a file, a client asks for a PPMDS server to create a file by passing the parent inode number and the entry name, and waits for the result. When the PPMDS server starts the transaction, it creates a state of the transaction in the local key-value store. During the transaction, access to key-value pairs may be done locally when it is stored in the local key-value store. When committing the transaction, it always changes the state of the transaction stored in the local key-value store. It is possible to eliminate many remote procedure calls in a transaction when a transaction starts at the server side.

4.2 Multiple Readers

For transactional key-value store, a key-value pair needs to be opened before accessing the key-value pair. If another transaction is already accessing the transactional key-value pair, a new transaction needs to wait to avoid the conflict. However, when the transaction only includes read accesses, it is possible to use an optimistic scheme. To allow multiple readers to access a key-value pair, we provide the "open for read" function. This function does not change the state of the transactional key-value pair. Instead, it copies the key and the version of a pair in a temporal space. When committing the transaction, it compares the current version and the copied version in the temporal space. If they are the same, there is no modification during the read transaction. The transaction is committed. However, if they are different, the transaction is forcibly aborted since the difference is caused by the write access of other transactions to the pair.

4.3 Shared Lock Mode

Multiple reader improves the concurrency for reading the same key-value pair by multiple transactions. However, there is a case such that the open for read causes the inconsistency in file system. When a directory removal transaction and a file creation transaction to the directory are concurrently processed, an orphan entry will be created as follows;

1. Directory removal transaction removes a list of distributed servers and checks whether the target directory is empty or not by range-based query, and it finds there is no entry.
2. File creation transaction obtains a list of distributed servers for the directory. At this time, the owner of the list of distributed servers is the directory removal transaction. Since it is not committed, there is still a list of distributed servers as an old value. The file creation transaction obtains the old value, and creates a new file entry.
3. When the file creation transaction is committed, it succeeds because the version is not changed.
4. Directory removal transaction removes an entry for the directory.
5. When the directory removal transaction is committed, it succeeds because the status is active.

In this case, nonempty directory will be removed and an orphan entry will be created. The problem here is the file creation transaction can read a list of distributed servers when it is active by the directory removal transaction.

To solve this issue, we introduce the shared lock mode for open for read. The shared lock mode prevents modification of the key-value pair to be read.

When a key-value pair is opened for read in the shared lock mode, the state of the transaction is checked. When it is committed or aborted, it succeeds to open for read in the shared lock mode. It obtains an up-to-date value, and saves the key and the version to a temporal space. When it is active, it fails to open the transactional key-value pair. It needs to wait for the conflict resolution, and starts again.

When committing the transaction for read in the shared lock mode, the copied version and the current version are compared. If it is not changed, the key-value pair read in the shared lock mode has not been updated. Thus, the

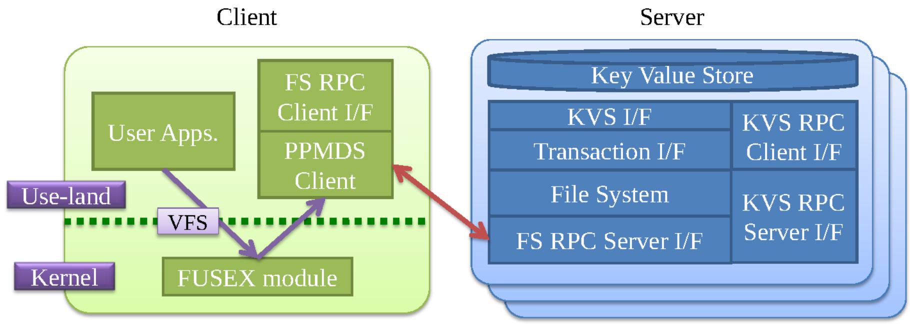

Figure 3: PPMDS Architecture

transaction succeeds. However, if it is changed, this means another transaction opens the transactional key-value pair and increments the version. In this case, the key-value pair read in the shared lock mode has been or will be updated. Thus, the transaction fails.

5 Implementation of PPMDS

Figure 3 illustrates the overview of the PPMDS architecture. PPMDS is multi-master type servers. Each PPMDS server has a local key-value store, and accepts client connections. The client interface is implemented using FUSE [Rath] framework.

5.1 PPMDS Server

Each server uses the Kyoto Cabinet [Hirabayashi] as a local key-value store. Since our design assumes the range-based query for directory listing, TreeDB is used for a key-value store for inode structures. Regarding a key-value store to store the state of transactions, StashDB is used for performance reason. For each key-value store, lock granularity is a row level. Both also support read-modify-write operations for a key-value pair, which enables to support atomic operations like compare-and-swap that is required by nonblocking transactions.

For message exchanges, MessagePack-RPC for C++ [Furuhashi] is used. Software stack in servers is shown in Figure 3. Each server has KVS RPC server interface to access local key-value stores, and FS RPC server interface for file system metadata operations.

The KVS RPC server interface is to access the local key-value store from remote clients and servers, and it implements the following atomic operations to support nonblocking transaction; *(version, value) = gets(key)* gets a value and a current version. *add(key, value)* adds a key-value pair only if the key does not exist. *replace(key, value)* updates a key-value pair only if the key already exists. *key = uadd(value)* generates a unique key and stores the given value, and returns the unique key. *cas(key, value, version)* updates a key-value pair only if the current version of the key-value pair is same as the specified version.

The FS RPC server interface currently supports file system metadata operations including initialization of the file system, file creation, file stat, file removal, directory creation, directory stat, and directory removal. File data operations such as read and write, are not supported.

The transaction interface supports the nonblocking distributed transactions. If a target key-value pair exists in the local key-value store, it accesses the local key-value store directly, but if not, it communicates to a remote PPMDS server via the KVS RPC client interface.

5.2 PPMDS Client

PPMDS client is implemented using FUSE low-level interface, fuse_lowlevel_ops. The following call-back functions are implemented; *init, destroy, lookup, getattr, opendir, readdir, releasedir, mkdir, rmdir, create, unlink.*

PPMDS manages inode entries by a parent inode number and an entry name. This means all operations require the parent inode number and an entry name. In the FUSE low-level interface, these information can be obtained by argument for all functions except the getattr. For the getattr operation, the parent inode number cannot be obtained. We extend the FUSE kernel module to obtain a parent inode number and the entry name. The following function is implemented;

```
fuse_req_get_idpair(req, &pino, &name)
```

req is a fuse_req_t structure used in a FUSE low-level interface. When fuse_req_get_idpair call succeeds, it returns the parent inode number by pino, and the entry name by name. The extended FUSE is called *FUSEX*.

6 Performance Evaluation

In this section, we evaluate the performance of the PPMDS regarding the metadata operations in a single directory, and the scalability.

6.1 Evaluation Environment

All cluster nodes have dual sockets of 2.40 GHz quad core Xeon E5620 processors with 24 GB of memory. 11 cluster nodes are used for clients, and 14 cluster nodes

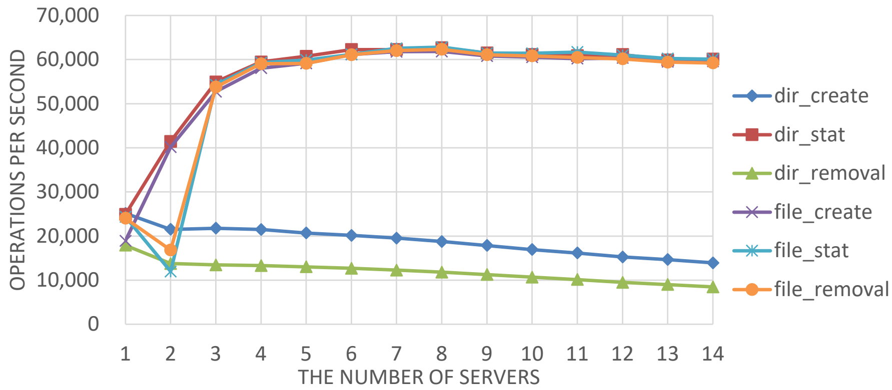

Figure 4: Performance of metadata operations using 88 client processes

are used for metadata servers. Client nodes are connected by Gigabit Ethernet, and metadata servers are connected by 10 Gigabit Ethernet.

For evaluation, we use mdtest benchmark [Torrez et al.] to measure the metadata performance. The mdtest is a parallel file system metadata benchmark in MPI. It creates files and directories under a specific directory structure, and stats their file system metadata, and removes them in parallel. It measures the operations per seconds. In this evaluation, each mdtest process creates 12,000 entries in a single shared directory. In the maximum configuration, 1,056,000 entries are created using 88 client processes in a single shared directory.

6.2 Parallel File System Metadata Operations in a Single Directory

Figure 4 shows the performance for parallel file system metadata operations in a single shared directory. The horizontal axis shows the number of PPMDS server nodes, and the vertical axis shows the number of operations per second for each operation. The total number of client processes is 88 using 11 client nodes. In each client node, 8 client processes are executed.

Regarding file operations such as file creation, file stat, and file removal, and also directory stat operation, it shows scalable performance up to 3 PPMDS servers, and achieves 62,000 operations per second. The performance improvement is about 2.58x from the one PPMDS server case. The reason why it does not scale in case of 4 or more PPMDS servers is that the number of client nodes is not enough to generate more request rate than 62,000 operations per second. In the next subsection, the PPMDS metadata server scalability is evaluated when changing the number of client nodes, which also supports this observation.

Regarding directory creation and directory removal, the metadata performance is not improved even when the number of servers increases. This is a design choice to prioritize the file performance rather than directory creation performance. When creating a directory, distributed transactions are posed to create a list of servers in all the related servers, which degrades the performance when the number of servers increases. When removing a directory, a distributed transaction needs to check entry existence, and then removes the directory. The performance result is along with the theory.

6.3 Scalability of Distributed PPMDS Metadata Servers

The performance for parallel file creations in a single shared directory is shown in Figure 5 when increasing the number of client processes. The horizontal axis shows the number of client processes, and the vertical axis shows the number of file creations per second. When using one PPMDS server, it shows the scalable performance up to 33 client processes and achieves up to 25,000 file creations per second, which is considered to be the maximum performance of one PPMDS server. Up to 33 client processes, the performance is limited by the client request rate. When using two PPMDS servers, it shows the scalable performance up to 44 client processes and achieves up to 40,000 file creations per second. On the other hand, when the number of PPMDS servers is 4 or more, it shows the similar scalable performance, which is limited by the client request rate. This means to evaluate the performance of PPMDS with 4 or more servers, more number of client processes are required.

7 Conclusion and Future Work

In these days, metadata accesses such as file creations or file opens can be a serious bottleneck in distributed file systems because of huge amount of files both in industry [Niazi et al. 2017] and science [Braam, Tatebe et al. 2010]. To cope with the issue, this paper proposed a scalable metadata server system, PPMDS. PPMDS is motivated from the Gfarm [Tatebe et al. 2010] distributed file system and is already integrated with an object storage [Takatsu et al. 2016] and forms a file system called PPFS [Takatsu et al. 2017].

For the scalable distributed metadata design, there are two major challenges; how to efficiently manage hierarchical namespace in parallel, and how to support all file system operations across multiple servers. For the hierarchical namespace, PPMDS manages inode entries using a parent inode number and an entry name as a key across multiple servers. This data structure is naturally distributed across multiple servers. To support all file system operations, PPMDS utilizes nonblocking distributed transactions for key-value stores.

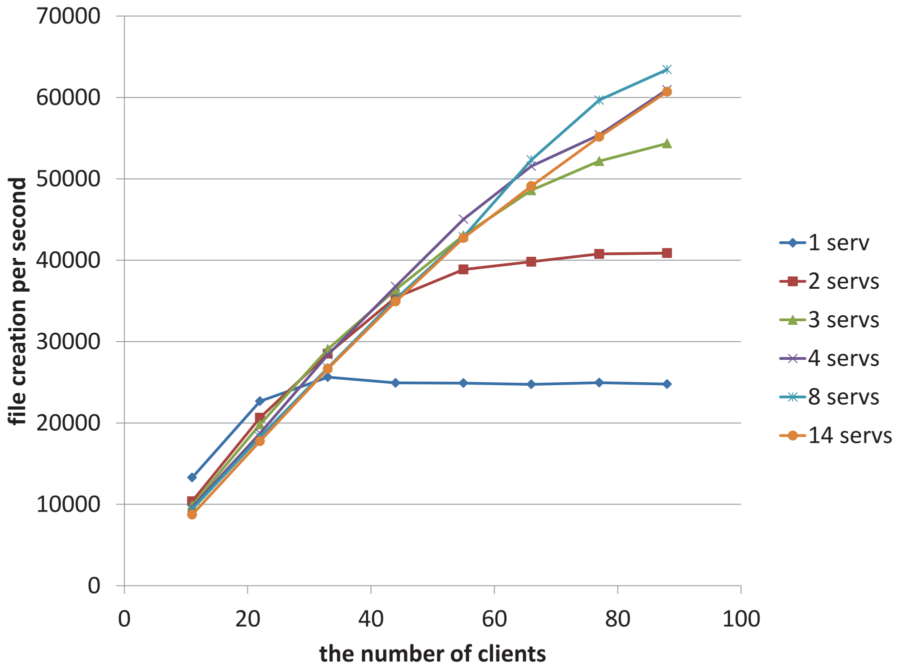

Figure 5: Parallel file creation performance in a single directory when increasing the number of clients

PPMDS improves the metadata access performance by introducing the server-side transaction processing, the open-for-read for multiple readers, and the shared lock mode scheme. These techniques reduce the number of remote procedure calls and avoid unnecessary blocking time.

In the evaluation, our implementation of PPMDS shows scalable performance up to 3 PPMDS servers, and achieves 62,000 operations per second. The performance improvement is about 2.58x from the one PPMDS server case.

Future work includes the performance evaluation in much larger scale environment using typical HPC applications.

Acknowledgment

This work is partially supported by the JSPS KAKENHI Grant Number 17H01748, JST CREST Grant Number JPMJCR1414, New Energy and Industrial Technology Development Organization (NEDO), and Fujitsu Laboratories.

References

[Braam] Braam, P. J.: "Lustre"; `http://www.lustre.org/`.

[Burrows 2006] Burrows, M.: "The Chubby lock service for loosely-coupled distributed systems"; Proc. 7th symp. on Operating systems design and implementation, OSDI '06 (2006) 335–350.

[Furuhashi] Furuhashi, S.: "MessagePack"; `http://msgpack.org/`.

[Ghemawat et al. 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T.: "The Google file system"; Proc. 19th ACM Symp. on Operating Systems Principles (2003) 20–43.

[Gray 1978] Gray, J.: "Notes on data base operating systems"; Operating Systems, An Advanced Course (1978) 393–481.

[Hadoop] Hadoop, A.: "Hadoop distributed file system"; `http://hadoop.apache.org/`.

[Herlihy et al. 2003] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N.: "Software transactional memory for dynamic-sized data structures"; Proc. twenty-second annual symp. on Principles of distributed computing, PODC '03 (2003) 92–101.

[Hirabayashi] Hirabayashi, M.: "Kyoto Cabinet"; `http://fallabs.com/kyotocabinet/`.

[Hiraga et al. 2018] Hiraga, K., Tatebe, O., and Kawashima, H.: "PPMDS: A distributed metadata server based on nonblocking transactions"; Proc. 2018 Fifth Int. Conf. on Social Networks Analysis, Management and Security (SNAMS), Valencia (2018) 202–208.

[Hunt et al. 2010] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B.: "ZooKeeper: wait-free coordination for internet-scale systems"; Proc. 2010 USENIX conf. on USENIX annual technical conf., USENIX ATC'10 (2010) 1–14.

[Kumazaki et al. 2011] Kumazaki, H., Tsumura, T., Saito, S., and Matsuo, H.: "Implementation of obstruction-free transaction on distributed key value store"; IPSJ SIG Tech. Rep. 2011-OS-118, 16 (2011) 1–7.

[Niazi et al. 2017] Niazi, S., Ismail, M., Haridi, S., Dowling, J., Grohsschmiedt, S., and Ronström, M.: "HopsFS: Scaling hierarchical file system metadata using NewSQL databases"; Proc. 15th USENIX Conf. on File and Storage Technologies (FAST 17), Santa Clara, CA (2017) 89–104.

[Patil and Gibson 2011] Patil, S. and Gibson, G.: "Scale and concurrency of GIGA+: File system directories with millions of files"; Proc. 9th USENIX Conf. on File and Storage Technologies (2011) 177–190.

[Rath] Rath, N.: "FUSE: Filesystem in Userspace"; `http://fuse.sourceforge.net/`.

[Ren et al. 2014] Ren, K., Zheng, Q., Patil, S., and Gibson, G.: "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion"; Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC '14 (2014)237–248.

[Samaras et al. 1993] Samaras, G., Britton, K., Citron, A., and Mohan, C.: "Two-phase commit optimizations and tradeoffs in the commercial environment"; ICDE (1993) 520–529.

[Schmuck and Haskin 2002] Schmuck, F. and Haskin, R.: "GPFS: A shared-disk file system for large computing clusters"; Proc. 1st USENIX Conf. on File and Storage Technologies, FAST '02 (2002) 1–14.

[Sinnamohideen et al. 2010] Sinnamohideen, S., Sambasivan, R. R., Hendricks, J., Liu, L., and Ganger, G. R.: "A transparently-scalable metadata service for the Ursa Minor storage system"; Proc. 2010 USENIX Conf. on USENIX Annual Technical Conf., USENIX ATC'10 (2010) 1–14.

[Takatsu et al. 2016] Takatsu, F., Hiraga, K., and Tatebe, O.: "Design of object storage using OpenNVM for high-performance distributed file system"; Journal of Information Processing, 24, 5 (2016) 824–833.

[Takatsu et al. 2017] Takatsu, F., Hiraga, K., and Tatebe, O.: "PPFS: A scale-out distributed file system for post-petascale systems"; Journal of Information Processing, 25 (2017) 438–447.

[Tatebe et al. 2010] Tatebe, O., Hiraga, K., and Soda, N.: "Gfarm grid file system"; *New Generation Computing*, 28, 3 (2010) 257–275.

[Torrez et al.] Torrez, A., Loewe, W., and Klundt, R.: "mdtest HPC benchmark"; `https://github.com/hpc/ior/`.

[Weil et al. 2006] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C.: "Ceph: a scalable, high-performance distributed file system"; Proc. 7th symp. on Operating systems design and implementation, OSDI '06 (2006) 307–320.

[Xiong et al. 2011] Xiong, J., Hu, Y., Li, G., Tang, R., and Fan, Z.: "Metadata distribution and consistency techniques for large-scale cluster file systems"; IEEE Trans. on Parallel and Distributed Systems, 22, 5 (2011) 803–816.

[Zheng et al. 2014] Zheng, Q., Ren, K., and Gibson, G.: "BatchFS: Scaling the file system control plane with client-funded metadata servers"; Proc. 9th Parallel Data Storage Workshop, PDSW '14 (2014) 1–6.

[Zheng et al. 2015] Zheng, Q., Ren, K., Gibson, G., Settlemyer, B. W., and Grider, G.: "DeltaFS: Exascale file systems scale better without dedicated servers"; Proc. 10th Parallel Data Storage Workshop, PDSW '15 (2015) 1–6.