A Highly Reliable Metadata Service for Large-Scale Distributed File Systems

Jiang Zhou, Yong Chen, Weiping Wang, Shuibing He, and Dan Meng

Abstract—Many massive data processing applications nowadays often need long, continuous, and uninterrupted data accesses. Distributed file systems are used as the back-end storage to provide the global namespace management and reliability guarantee. Due to increasing hardware failures and software issues with the growing system scale, metadata service reliability has become a critical issue as it has a direct impact on file and directory operations. Existing metadata management mechanisms can provide fault tolerance capability to some level but are inadequate. They often have limitations in system availability, state consistence, and performance overhead and lack an effective mechanism to offer metadata reliability. This paper introduces a novel highly reliable metadata service to address these issues in large-scale file systems. Different from traditional strategies, this proposed reliable metadata service adopts a new active-standby architecture for fault tolerance and uses a holistic approach to improve file system availability. A new shared storage pool (SSP) is designed for transparent metadata synchronization and replication between active and standby servers. Based on the SSP, a new policy called multiple actives multiple standbys (MAMS) is presented to perform metadata service recovery in case of failures. A new global state recovery strategy and a smart client fault tolerance mechanism are achieved to maintain the continuity of metadata service. We have implemented such highly reliable metadata service in a prototype file system CFS (Clover file system) and conducted extensive tests to evaluate it. Experimental results confirm that it can significantly improve file system reliability with fast failover under different failure scenarios while having negligible influence on performance. Compared with typical reliability designs in Hadoop Avatar, Hadoop HA, and Boom-FS file systems, the mean-time-to-recovery (MTTR) with the highly reliable metadata service was reduced by 80.23, 65.46 and 28.13 percent, respectively.

Index Terms—Distributed file systems, metadata service, metadata reliability, fault tolerance, shared metadata storage

1 Introduction

With the arrival of the big data era, massive data processing has been widely used for extracting useful knowledge from a large amount of datasets. Most applications, including offline and online data analysis, often need long, continuous, and uninterrupted data accesses. Mass data storage has become a key building block for large dataset processing. Distributed file systems have attracted intensive attention in recent years and are being actively investigated to meet new demands for the management of massive datasets and a variety of data structures.

The metadata management is a critical component of distributed file systems and plays a key role in terms of their scalability, reliability, and availability. There are two metadata management mechanisms in general: a *centralized* mechanism and a *decentralized* mechanism. The *centralized* metadata management, including Lustre [1], PVFS [2], GFS [3], and HDFS [4], has one metadata server to organize all metadata of files and directories. It greatly simplifies the design and implementation of distributed file systems but can easily lead to a single point of failure: once the metadata server crashes, the file system will not be available. The *decentralized* metadata management, including Ceph [5], Ursa Minor [6], and HDFS Federation [7], uses a group of metadata servers to manage the global namespace. It improves file system scalability and is the trend for future distributed file systems. However, a decentralized metadata management mechanism presents challenges in metadata operation performance and in the case of multiple metadata server failures. In this research, we focus on these challenges and introduce a highly reliable metadata management policy for multiple metadata services. We present its design, implementation, and evaluation results in this paper.

The reliability of file system metadata management has never been so important. Advances in large-scale cluster and high performance computing systems enable effective and rapid mass data processing. However, larger systems generally have more processing components and other elements, which increase the overall system failure rate. Although the mean-time-between-failures (MTBF) for an individual component may be high, the system reliability can decrease significantly in a large-scale system. Experience in Facebook indicates that modern data centers are rife with hardware failures, in which the large scale of deployment both ensures a non-trivial fault incidence rate and complicates the localization of these faults [8]. Another example is the MTBF of IBM Blue Gene/L machine, which is built with

- *J. Zhou, W. Wang, and D. Meng are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China. E-mail: {zhoujiang, wangweiping, mengdan}@iie.ac.cn.*
- *Y. Chen is with the Department of Computer Science, Texas Tech University, Lubbock, TX 79401. E-mail: yong.chen@ttu.edu.*
- *S. He is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310058, China. E-mail: heshuibing@zju.edu.cn.*

Manuscript received 26 Jan. 2019; revised 4 July 2019; accepted 6 Aug. 2019. Date of publication 26 Aug. 2019; date of current version 26 Dec. 2019. (Corresponding author: Jiang Zhou.) Recommended for acceptance by W. Yu. Digital Object Identifier no. 10.1109/TPDS.2019.2937492

131,000 processors and is estimated to be below 7 hours [9]. In fact, recent studies indicate that servers tend to crash with 2-4 percent failure rate, 1-5 percent of disk drives die, and 2 percent memory errors occur per year [10], [11]. In an extreme-scale system with more than hundreds of thousands of nodes (part of them are designated as metadata servers), the MTBF is expected to fall below tens of minutes, and a node/server error can occur every day on average [12].

In addition to increased hardware failures, many systems often encounter software upgrades, configuration changes, installation of new components, and planned or unplanned downtime [13]. As a typical distributed framework, most Hadoop clusters will perform major upgrades every quarter and relevant software stack patch at any time. A metadata service failure can have a significant impact. For example, it will lead to tens of thousands of file system request failures at the Facebook real-time system [14] if the metadata server suspends. Due to the increasing likelihood of hardware failures or software issues, the metadata service can be interrupted and lead to the entire file system unavailability, which has a catastrophic impact on big data applications.

One common way for maintaining metadata reliability in file systems is to depend on logs or checkpoints [15]. The log-structured file systems [16] learn from the database transaction processing [17] for metadata recovery from failures. Some local file systems [15], [18] or distributed file systems [19], [20] use log-based design for system reliability. Metadata modifications are first written to a sequential journal on disks before being applied to operations. When in recovery, the journals are read in memory to reconstruct the file system namespace by replaying or rolling back logs. Although this design improves the reliability of metadata service, it has a critical limitation: whenever the server is down, the file system is unavailable until it is fully recovered from logs. Besides, when the server stores metadata on persistent devices, it is restricted to read and write operations of local disks. It is important to tolerate the failure of metadata service for satisfying such applications whose uptime requirement is 24×7 in mass data analysis and processing.

To provide a highly reliable metadata service, the primary-backup strategy is often used in distributed systems such as Lustre [1] and GFS [3]. If the primary server crashes, a backup one will take over its role and continue to respond to clients. However, there is a service downtime before the backup finishes metadata recovery and replaces the primary. For example, the largest HDFS cluster in Facebook with 150 million files takes about 20 minutes for failover [14]. To achieve seamless service switching, numerous strategies [7], [21], [22] adopt hot standby for the primary. It reduces the recovery time, but still needs dozens of seconds and minutes for recovery and still has the problem of metadata state inconsistency. Server state replication [23], [24] is another way to improve metadata reliability. All metadata modifications are replicated in multiple standbys when an active is running. One of the standbys will provide the active service in case of failures. It enhances the file system reliability. Current strategies, however, suffer three limitations. First, they have influence and affect the performance of normal metadata operations. Second, they require additional failover time for system state recovery. Third, they lack an effective method to tolerate failures for multiple metadata services in distributed storage systems.

To address these aforementioned limitations, we propose a novel, highly reliable metadata service to largely improve metedata reliability for large-scale distributed file systems. In contrast to existing systems and solutions, the contribution of this study is four-fold. First, we introduce a novel shared storage pool (SSP) design for metadata persistence. The SSP works as a virtual store for reliable metadata sharing and storage. It is built upon existing metadata servers and internally adopts an optimized two-phase commit protocol (2PC) for metadata/journal synchronization and replication [25], which not only provides consistency guarantee among servers but also significantly reduces the overhead on metadata operations. The shared storage pool is transparent for metadata servers, without requiring any additional hardware support, which otherwise could increase the cost and decrease the flexibility and likelihood of adoption. Experimental results show that SSP improves the reliability of metadata service for recovery.

Second, we propose a new active-standby policy called *multiple actives multiple standbys (MAMS)*, to perform metadata service recovery in case of failures. The MAMS policy has two significant advantages. One is that it can tolerate various failures and significantly improve metadata reliability in a large-scale file system. To achieve that, MAMS divides metadata servers into different replica groups and maintains more than one standby server for recovery in each group. When the active server fails, a new one will be elected from standbys to replace the active and recover metadata service. The other advantage is that MAMS can reduce the overhead of recovery with little influence on file system performance. It is based on the shared storage pool for metadata consistency between active and standby servers. By using an elaborate interactive state transition among servers, it achieves seamless service switching in the form of hot standby. The MAMS policy also supports adding backup nodes at runtime dynamically. Performance results show that MAMS achieved a fast failover within milliseconds for metadata operations.

Third, we present a new global state recovery mechanism for distributed file systems. When multiple points of failure occur, some metadata servers may not be recovered even by the MAMS policy, e.g., the operating system crashes or a backup node fails. It needs to restart all servers and perform failover for the entire file system. The mechanism assigns unified checkpoint flags in the journals of each metadata server. When the file system is resumed, the server read journals to construct the namespace state till it meets the unique flag. It ensures that the file system recovers to a consistent state of previous time after recovery.

Fourth, we present a novel smart-client protocol for transparent fault tolerance. Unlike traditional manual service switching or virtual IP approaches, the client can automatically find and connect to a new one when the metadata server fails. Combining duplicate journal detection with automatic reconnection mechanism, the client protocol helps the MAMS policy to achieve quick failover with metadata consistency guarantee.

The rest of this paper is organized as follows. Section 2 outlines the related work. Section 3 describes the system

design of the highly reliable active-standby cluster. Section 4 presents our highly reliable metadata service. Section 5 evaluates the performance and characteristics of the service, with comparisons to widely-used highly reliable metadata management strategies. Section 6 concludes this paper and outlines our future studies.

2 Related Work

2.1 Distributed Metadata Storage

When providing metadata management, the file system needs to persist metadata operations/logs to storage device such as disks. It saves the namespace state and prevents metadata information from becoming unavailable. In distributed file systems, the metadata server will also interact with each other for state synchronization or state backup. To ensure file system reliability, metadata synchronization is often bound with replication, which can keep both metadata redundancy and consistence.

Metadata Synchronization. Several known studies exist and mainly focus on the synchronization strategy to keep consistent states among servers. An easy way is for the metadata server to transmit metadata modifications to other servers directly, such as BackupNode in HDFS [4]. It does not need to access shared files and only incurs network communication overhead. This method can improve the synchronization performance but cannot ensure state consistence among servers if errors occur during metadata transmission. To keep metadata consistent, several distributed protocols can be used for coordination. The two-phase commit protocol (2PC) [25] and variants [26] are atomic commit protocols to achieve consistency for distributed operations. They can be used to maintain consistent states among servers when synchronizing metadata. TAPIR [27] is a transactional application protocol that makes clients participate in 2PC process to reduce the number of round-trips to storage servers. Some modified protocols [28] combine the 2PC protocol with metadata processing to perform cross-metadata server operations. Although they ensure metadata state consistency, they can lead to additional overhead on normal metadata operations. Shared storage [29] is an efficient method for metadata synchronization in which the metadata server writes metadata operations into the shared files and the other server reads them and updates its namespace state on time. NFS [30] allows users to access the shared resources transparently. The metadata servers mount the remote file system in a local machine and perform writing and reading for the files simultaneously. BookKeeper [31] is a distributed log storage system. It is based on the message queue for communication and writes ledgers into multiple bookie nodes. BookKeeper also considers fault tolerance by replicating each ledger entry. It can wait for results asynchronously in which it needs to reselect a bookie for writing the fault replica when some node crashes. BookKeeper can store journals effectively, but it adds additional steps for shared file access and increase the synchronization overhead.

Metadata Replication. There is a long history of distributed storage systems that employ replication to improve system reliability [32], [33], [34], [35], [36]. These solutions provide high availability and consistent semantics to handle system failures. However, they either increase the operation cost or need additional hardware support, which reduces the flexibility and likelihood of adoption. To enhance system reliability, numerous strategies adopt replicated state transition to achieve highly reliable services [37], [38]. TidyFS [39] maintains multiple metadata servers as a replicated component for the centralized metadata server. It leverages the Autopilot Replicated State Library [40] and Paxos [41] protocol to solve consensus in a network of unreliable processors. Boom-FS [23] implements a data-centric analytics stack, which is compatible with Hadoop interfaces. To achieve reliability, it adopts a globally-consistent distributed log to guarantee a total ordering over events affecting replicated states. Both of them use the Paxos protocol to maintain consistency among servers and provide a reliable mechanism by the replicated state machine approach. The operation performance, however, is affected by centralized repair action decision and state transition, which leads to additional failover time.

Compared with these methods discussed above, we propose a new shared storage pool (SSP) for distributed metadata storage in distributed file systems. The shared storage pool is designed as a virtual store for metadata persistence (journal and image files). It is built upon existing metadata servers, without any additional hardware support. Through an optimized 2PC protocol, the shared storage pool can achieve efficient metadata synchronization and replication between the active and standby servers. It not only improves metadata service for recovery but also reduces the overhead for normal metadata operations.

2.2 Reliable Metadata Service

Metadata reliability strategy is widely adopted in large-scale storage systems. It helps keep the continuity of metadata service and maintains the state consistency of file systems in case of failures.

Traditional Primary/Backup Strategy. To support continuous service for metadata operations, the primary/backup strategy is often used, such as in PVFS [2], Lustre [1], HARP [42], GFS [3], HDFS [4] and etc. [43]. It achieves fault tolerance by using the backup server to take over as the primary if the latter crashes. When the active server provides service, the backup server keeps updated status with it for state recovery. It has little extra overhead on normal operations but easily leads to incorrect states between the primary and backup without consistency guarantee. Furthermore, as it requires the procedure of restarting and reconnection in the backup server, it takes a long time for failover. In contrast, our proposed MAMS policy provides an automatic hot standby, which significantly reduces the overhead for recovery.

Hot Standby Strategy. The hot standby strategy provides seamless service switching for some systems to speed up their failover recovery [7], [14], [22]. AvatarNode [14] at Facebook is designed for a realtime file system that supports online applications and requirements. In order to construct file locations, the datanodes talk to both active and standby metadata servers. As the standby server keeps the same state with the active server on the fly, it can take over quickly. Hadoop HA [7] employs the Quorum Journal Manager (QJM) to synchronize server states between the active and standby. They can achieve automatic fault tolerance but still take some time for recovery and have influence on normal metadata performance. Wang et al. [22] have proposed a primary-slaves topology to enable Hadoop high availability.

The mechanism consists of one primary critical node and several slave nodes for removing a single point of failure. It spends about three times of the response delay in metadata operations. Different from above methods, our MAMS policy achieves an automatic hot standby for multiple metadata service in distributed file systems. Each metadata server has one or more standbys that maintain the same namespace state with it. If one metadata server crashes, a new active will be elected from other standbys to replace it and achieve fast service takeover.

Reliability Management Strategy. Symmetric or asymmetric active/active models are also used to improve file system reliability [44]. In the symmetric strategy [45], more than one dedicated server provides the shared global state with the virtual synchrony technique. With a fast delivery protocol, service node failures do not cause a failover to a backup and there is no disruption of service or loss of service state. Asymmetric strategy [46] comprises $n+1$ and $n+m$ configurations with n active nodes and 1 or m standby nodes. The standby servers monitor all active servers and perform failover when the outage is detected. As client requests are distributed across multiple active servers, these strategies will result in a performance decrease in metadata processing and additional overhead for response time. In contrast, our proposed MAMS policy adopts distributed coordination and achieves faster, transparent fault tolerance within milliseconds while delivering high performance for metadata operations.

Zookeeper [47] provides a high-performance coordination service for distributed applications. However, Zookeeper mainly provides a simple set of primitives that distributed applications can build upon to implement higher lever services. We use Zookeeper to coordinate implementation of the proposed MAMS policy, such as monitoring nodes, triggering events and maintaining server states. Based on Zookeeper, the MAMS adopts a Paxos-like election algorithm that ensures only one active server is elected each time in a replica group. Comparing with consensus algorithms Zab [48] or Raft [49], it uses two distributed protocols for fault tolerance and recovery. One is the failover protocol that achieves automatic state transition for metadata servers between three different states. The other is the renewing protocol that can recover backup nodes from obsolete state and allow dynamically adding backup nodes. Global state recovery and smart-client fault tolerance are also achieved for seamless failover. Evaluations have confirmed the efficiency of such highly reliable metadata service.

3 System Design

The goal of this research study is to achieve a highly reliable metadata service to improve metadata reliability for large-scale file systems. In this section, first we discuss the challenges of achieving that, then we introduce the overall system design. In the next section, we present the design and implementation in detail.

3.1 Challenges

To achieve the research objective, a highly reliable metadata service faces the below challenges:

- Metadata consistency. To achieve hot standby, the backup node keeps a consistent namespace state with the active server. Paxos-based methods enhance the consistency but reduce the performance. In most cases, it is a trade-off between the metadata operations and metadata consistency, e.g., offering either strong or eventually consistency guarantees [50], [51]. Thus, an effective method is required to synchronize and replicate metadata modifications among servers.
- IO Fencing. During the failover situation, it is important that only the active server has the obligation to update the shared metadata information. The backup node performs read operations under failure free cases. When it needs to elect a new active server, the previous one is replaced but might not be isolated immediately and may continue to perform metadata modifications. IO fencing is required to ensure that the obsolete server does not update the shared journals and to prevent fault pervasion.
- Split-brain problem. Highly-available clusters often use the heartbeat mechanism to monitor each node and the health status of the whole system. The split brain scenario may happen when some errors occur, such as transient network failures. At this time, each backup node may believe it is the only one running and wants to take over the active server. The cluster could enter an uncertain state for clients, with the risk of generating new errors.
- Performance overhead. When using a highly reliable mechanism, it will result in additional overhead for metadata operations under failure or failure-free states. Low performance cost, including metadata state synchronization, fault tolerance and failure recovery, should be considered when designing a highly reliable policy.
- System scalability. The scalability includes two aspects: one is the ability of a file system to scale with multiple metadata servers to manage a global namespace. The other is the ability of a reliable policy to support scaling backup nodes. When using a highly-available cluster for system reliability, there is a risk of server failures and will reduce the number of available servers. It is necessary to add backup nodes dynamically for improving the reliability of the entire storage system.

3.2 System Architecture

In a typical distributed file system, multiple metadata servers are deployed to manage a global namespace. Partition-based methods [5], [7], [52] are often adopted for metadata management and metadata distribution. Based on the partition, each metadata server manages a portion of the namespace and responds to clients independently. The applications perform metadata and I/O requests through client interfaces for file system operations.

To support an uninterrupted service, our highly reliable metadata service adopts a novel active-standby architecture to improve system availability. As shown in Fig. 1, each metadata server (MDS) in the cluster is deployed with multiple dedicated backup nodes for reliability. For instance, the server MDS_1 has several backup nodes. The backup node can have two states, a standby state and a junior state, discussed in detail below. For each metadata server, a

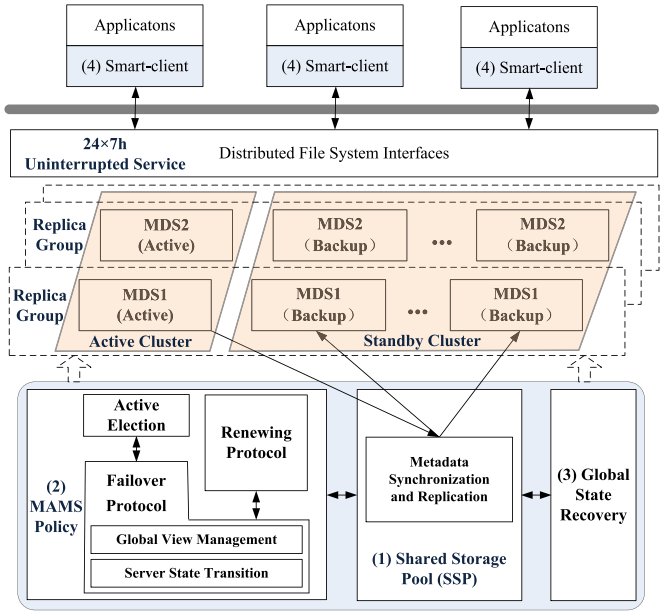


Fig. 1. System architecture overview with highly reliable metadata service.

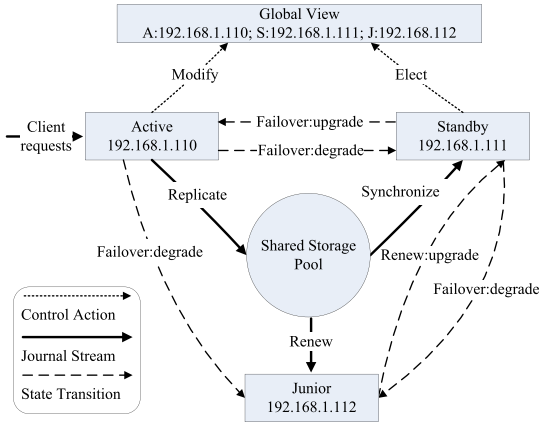


Fig. 2. Server state transition in the replica group. There are three states for a metadata server: active, standby, and junior. The state of servers may be switched to each other under different conditions.

replica group is constructed with one active and two or more standby/junior nodes. By server state replication, the standby nodes can keep the same states with the active and take over its role in the event of failures. Then the active and backup metadata servers constitute an active-standby cluster to provide file system service for applications.

The use of standby cluster can improve metadata reliability, but may introduce additional cost and require additional maintenance. In fact, it is always a trade-off between metadata server redundancy, performance, and reliability. First, we use multiple metadata servers to provide file system service, which can be highly scalable and beneficial for performance [52]. Second, adding standbys can improve metadata reliability. Our theoretical analysis shows that the mean time to failure (MTTF) is improved by 3 magnitudes when adding each additional backup node for one active in the replica group (see details in Section 5.1). Third, our evaluations show that the metadata performance is gradually decreased when adding standbys but the influence is minor (see details in Section 5.3). Thus, it is worthy to use a standby cluster to achieve reliability for multiple metadata service.

With the active-standby architecture, our highly reliable service adopts four collaborative and tightly connected modules to improve file system availability, as illustrated in Fig. 1. The first module is a virtual shared storage pool (SSP) [52], [53], which distributes the responsibility of managing different metadata partitions among different groups of servers. The SSP provides a function for metadata synchronization and replication between the active and standby servers with little performance overhead. It will be discussed in detial in Section 4.1. The second module provides a MAMS (multiple actives multiple standbys) policy, which is responsible for fault tolerance and service switching if errors occur at metadata servers. When the active server fails, a new one will be elected from backup nodes, which is a synchronization step similar to the Paxos leader election phase. A global view management (which maintains server states) and two distributed protocols, failover protocol and renewing protocol, are used in MAMS for server state transition. We will discuss it in detail in Section 4.2. Third, a global state recovery mechanism is designed to recover the entire file system to a previous consistent state. This mechanism will be discussed in detail in Section 4.3. Fourth, a smart-client component is designed to achieve transparent fault tolerance and to enable client applications to better leverage the design of the metadata storage system. We will discuss it in Section 4.4.

3.3 Server State Transition

For each replica group in the active-standby cluster, the metadata server starts up in different states according to a global view. The global view keeps all server states in the active-standby cluster and helps achieve server state transition. We use Zookeeper [47] to maintain the global view, monitor server states, and fundamentally coordinate server state transition in the MAMS policy. The metadata server acts as one role and is transitioned to other states upon different conditions. The state of server is defined as follows:

- Active. An active server receives client requests and provides metadata service for the namespace partition it manages. In any replica group, one and only one of servers is in an active state at any time.
- Standby. A standby server is a backup node, which simply keeps an up-to-date namespace state with the active at any time (using strong consistency). It does not provide metadata service but can take over as the active server in the event of failures.
- Junior. A junior server is a type of backup node, which is in an intermediate status and cannot provide hot standby. It does not synchronize metadata with the active server in real time but will finally recover state, or update to standby if there is no longer strongly consistent with regard to the active (using eventual consistency). It can be a server which restarts after a failure, or is a newly added backup node.

Under different conditions, the state of servers may be switched to each other. Fig. 2 shows a diagram of server state transition in the replica group. It can be seen that there are three servers in different states: one is an active server with the IP address 192.168.1.110, one is a standby server with the IP address 192.168.1.111, and another one is a

junior server with the IP address 192.168.1.112. As mentioned above, distributed protocols are responsible for the state transition. The failover protocol performs upgrading or degrading between the active and standby servers. It also degrades them to the junior state when necessary. The renewing protocol is adopted to upgrade the junior to a standby. Based on the SSP, the active server synchronizes and replicates journals to standby for keeping the state consistent. It reduces additional overhead with little influence on normal metadata operations. A monotonically increasing serial number (sn) is assigned by the active when it writes journals. Each batch of log records is described with the pair $<sn, groupid>$. By subtracting the values between two sn, the junior can retrieve missing metadata from the SSP or the active server for upgrading to the standby state.

As the distributed metadata service decouples data and metadata management, file contents are split into blocks and replicated in the data server cluster. Block locations are periodically reported via heartbeat [4] to both the active and standby nodes by data servers. It means that the standby node has the up-to-date file locations and can achieve a hot standby for the active server.

4 Highly Reliable Metadata Service

We introduce the design and implementation details of our newly proposed highly reliable metadata service, which provides consensus guarantee for fault tolerance. Algorithm 1 lists key properties of the highly reliable metadata service, where each element is discussed in detail over the rest of this section.

Algorithm 1. The Highly Reliable Metadata Service Guarantees that all these Properties are Met. The Section Numbers Indicate where each Property is Discussed

```
1  while receive client requests;
2  do
3      Metadata synchronization safety check: the active
       synchronizes metadata to standbys via the SSP.
       Section 4.1.2
4    if active fails;
5     then
6      Election safety check: at most one active can be elected
       in a given term. Section 4.2.1
7      State transition safety check: a new active will take
       over the service, other metadata servers change to
       states of standby or junior. Section 4.2.2
8      Metadata consistency check: server states change cor-
       responding to the SSP, and only the new active can
       write journals to standbys. Section 4.2.2
9      Server state consistency check: the junior will update
       to standby based on the SSP. Section 4.2.3
10     Client state safety check: the client reconnects to the
       new active and resends requests for failed operations;
       the new active distinguishes duplicated requests for
       fault tolerance handling. Section 4.4
11   end
12 end
13 Global state safety check: recover the entire file system to a
     previous, consistent state in case of multiple points of fail-
     ures. Section 4.3
```

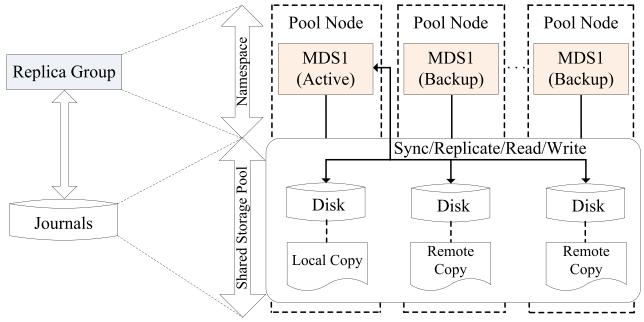


Fig. 3. Shared storage pool (SSP) design.

4.1 Shared Storage Pool

4.1.1 Shared Storage Pool Design

To achieve a highly reliable metadata service, a novel shared storage pool (SSP) is proposed in our approach. The SSP is an intermediate component that offers support for metadata reliability. It has two major roles. First, the SSP provides metadata synchronization between the active and standby servers. To achieve a hot standby and service takeover, standby servers need to synchronize metadata modifications from the active and keep consistent state with it. The SSP uses an internal protocol for metadata synchronization between the active and standbys. Second, the SSP provides persistent metadata storage. Besides maintaining namespace in memory, the metadata server has to store journals in disks for recovery. Otherwise, the file system state will be lost if the metadata server fails. Different from local disk storage, the SSP provides shared metadata storage where the active makes replication for journals and distributes them to standbys in each replica group.

Each active has two types of journals that are persistently stored, including a namespace image and one or more journal files. When the active synchronizes metadata to standbys, it writes logs into journal files in one local copy and replicates them in standbys simultaneously. In addition to journal files, the entire namespace in memory is stored periodically, as an image file. Like HDFS [4], it includes the inode data and the list of blocks belonging to each file in the file system. The image file is replicated between the active and standbys in a pipeline fashion [4]. Thus, the SSP can help to improve metadata redundancy and locality when metadata servers load journals for recovery.

Fig. 3 depicts the design of the shared storage pool. As shown in the figure, the active and multiple standby servers in each replica group form a pool of nodes as SSP. For each metadata operation, the active synchronizes and replicates modifications to its standbys through the SSP. As metadata servers are reused in the SSP, their states will change correspondingly if active-standby state transition happens. For instance, if an active is changed to standby due to failures, it will not synchronize metadata to others any more but just wait to receive metadata from the new active. Together with the MAMS policy and client fault tolerance handling, the SSP ensures eventual metadata consistency during server state transition (discussed below).

4.1.2 Metadata Synchronization and Replication

This section introduces the journal synchronization and replication protocol in SSP. Through the protocol, the active

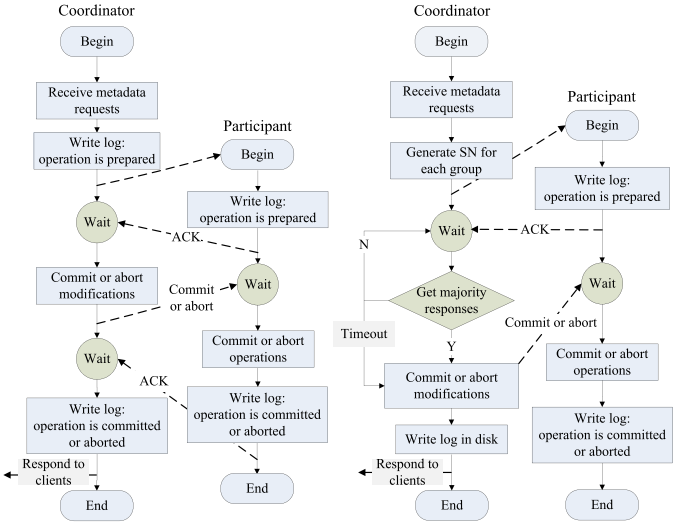


Fig. 4. Journal synchronization and replication protocol in SSP.

synchronizes state to its standbys and stores journals in a distributed way. To improve the performance, the active aggregates metadata modifications as a batch of journals and synchronizes them to standbys. As data servers periodically report block locations, the standby nodes can achieve a hot standby for the active when failures occur. To reduce the performance overhead on metadata operations, we design a distributed protocol based on the two-phase commit protocol (2PC) [25] for metadata synchronization and replication. For comparison, we first discuss the 2PC algorithm briefly and then introduce our protocol.

Fig. 4 shows the process of metadata synchronization and replication with the optimized 2PC protocol in SSP [53]. The 2PC protocol is a popular atomic commitment protocol used for distributed operations. Fig. 4a shows the workflow of using existing 2PC protocol method in SSP, which includes the pool node who initiates the write operation (coordinator) and the pool node who receives log records (participant). It is divided into two phases, a preparation phase and a commitment phase. In the preparation phase, the coordinator receives metadata requests from the client, writes the log with "Prepare" label in local disk, and transmits them to each participant. The participant receives journals and writes the "Prepare" log. In response, the participant then sends an "ACK" message to the coordinator to confirm the operation. The coordinator waits and collects ACK messages from participants in the commitment phase. When receiving all ACK messages, the coordinator commits modifications in memory and sends " Commit" messages to participants for execution. After all participants complete operations and write commit logs, they send ACK messages to the coordinator. When receiving all ACK messages, the coordinator write the commit log and responds to the client. During the procedure, the operation will be aborted if some participants send "Abort" messages or other errors occur.

With the 2PC protocol for metadata synchronization, it can ensure metadata consistency among metadata servers. However, it needs four times of the message communication between the coordinator and participant, which can significantly affect the performance of metadata operations. Besides, the 2PC protocol lacks the critical fault tolerance mechanism in the commitment phase for metadata synchronization.

To address these issues, we introduce an optimized 2PC protocol for metadata synchronization and replication in SSP. It acts like quorum-based Viewstamped Replication [54] or Paxos protocol [41], but is specifically designed and optimized for metadata operations and works together with other components for metadata reliability. The process is illustrated in Fig. 4b. First, the coordinator receives client requests and aggregates them in a group. For each group, the coordinator assigns a monotonically increasing serial number (sn) for operation ordering and fault tolerance. It is described with the pair $<sn, groupid>$. When a group of log records are prepared, the coordinator sends them to all participants. Second, each participant receives metadata modifications, writes the "Prepare" log in local disk, and send an ACK message to the coordinator. At last, when the coordinator receives a majority of responses (more than half), it performs operations in memory and responds to the client. The coordinator can use the sn to check whether it receives correct responses from participants. After that, the coordinator notifies participants that have returned ACK messages to commit and write logs. As the coordinator considers the synchronization operation has been finished, it deals with subsequent responses from participants via asynchronous threads. If the coordinator finds ACK messages timeout, it does not send journals to related participants any more. Different from the existing 2PC protocol, the coordinator concludes the operation as a success if more than half of participants return ACK messages. During this process, the coordinator does not need to write journals twice. It stores the commit or abort logs in local disk directly. Supposing N is the number of replicas, our policy can tolerate $(N-1)/2$ failures and continue to work normally. As the journal file is written sequentially and read when the file system restarts or recovers, we relax the metadata consistency model for journal synchronization and replication. It does not maintain the same context for replicas at writing operations but ensures the eventual consistency when metadata read operations happen.

Correctness. Two ways are provided for fault tolerance and eventual consistency guarantees in the SSP. First, if the standby fails during metadata synchronization and replication, it will change to junior state and do not receive metadata modifications. As time passes, the junior will have an obsolete metadata state compared to the active. To address that, a distributed protocol described in Section 4.2.3 updates junior to standby. After recovery, the standby will receive metadata update from the active and act as standby node in the SSP again. Second, if the active fails during the protocol process, a client fault tolerance mechanism will cooperate with the MAMS policy for metadata consistency (as seen in Sections 4.2.2 and 4.4). For service switching in the reliable paradigm, the new active may flush the last group of journals to standbys or the client may resend failed requests for dropped operations. At this time, duplicated metadata modifications will be detected at metadata server sides to achieve correct file system semantics. The MAMS policy will ensure server states consistency during active-standby transition, as well as their states in the SSP.

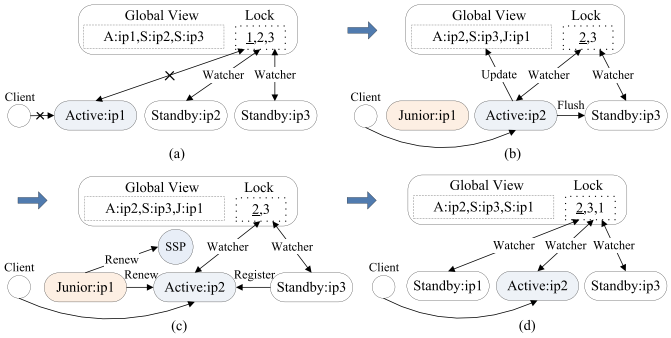


Fig. 5. The main procedure of active-standby transition.

4.2 The MAMS Policy

Based on the SSP, we propose a new MAMS (multiple actives multiple standbys) policy that uses the active-standby cluster for fault tolerance. By maintaining a prepared state replication and automatic state transition, the MAMS policy achieves quick recovery in the form of hot standby. We introduce the detailed design and implementation in this section.

4.2.1 Active Election

As the MAMS policy uses more than one standby for fault tolerance, it needs to elect a new one when the active server crashes. Based on Zookeeper, MAMS use a Paxos-like algorithm [41] that ensures only one active server is elected each time. Combining the algorithm with a global view management, MAMS avoids the scenarios of split-brain. The process of active election is like distributed lock management. When the active crashes, each standby tries to obtain a distributed lock periodically until it either succeeds or encounters a failure (e.g., timeout). Active election can also be achieved by comparing log sn values. It ensures the continuity of metadata service even if no standbys are in the global view. Algorithm 2 describes the active election algorithm in a replica group.

Algorithm 2. Active Election in the Replica Group

```
1  stdby = the number of standby nodes;
2  if active has errors or no active in the global view;
3    then
4      if stdby > 0;
5       then
6         while no active is elected;
7         do
8            each standby generates a random number in the
              global view;
9            the standby S_i with the largest random number
              obtains the lock and upgrade to active;
10        end
11     end
12     else
13        while no active is elected;
14        do
15           select the junior J_i with maximum sn;
16           J_i obtains the lock and takes over as the active;
17        end
18   end
19 end
```

4.2.2 Active-Standby State Transition

The MAMS achieves an automatic active-standby switch for fault tolerance. Combined with the heart beat from the node monitor, the MAMS uses an event-driven mechanism to trigger active election and state transition in case of failures. Each server has three event watchers on the global view: one is on itself, one is on the active server and another is on the distributed lock. Any error will trigger them to modify the global view and result in two situations: the active state is changed, which makes the active server lose the lock, and an election process is launched, or other state transition between standby and junior. It ensures that no processes can obtain the distributed lock before the active loses it. The main procedure of active-standby transition for failover protocol is depicted in Fig. 5, in which the underlined number means the server with the same IP sequence that has granted the lock.

When the active server has detected failures, it stops providing service and no longer responds to clients, as shown in Fig. 5a. In this example, the active is directly degraded to junior. The role of active will also change in the SSP, which does not synchronize metadata to standbys. However, some obsolete data, such as buffered journals in the active, may be flushed to standbys. It does not matter because the standby only receives and responds for metadata that come from the active. As the global view will be modified immediately and the event is triggered whenever there are changes in server states, the scenario of two metadata servers accessing the same journal file simultaneously does not exist, which achieves the function of IO fencing. In some cases, the standbys have fine states in the global view, but network failures may happen between them. The watcher will not announce any error, but the MAMS can handle this failure in the renewing procedure as discussed below.

Once a standby server obtains lock successfully, it holds the lock and prepares for state transition. Events are triggered to notify others to stop competing, which will reduce unnecessary actions for election. The elected standby then does not receive any journals from the active and awaits an opportunity to switch. If there are no pending or processing operations, it will launch upgrade immediately. Otherwise, the elected standby applies them to its own namespace and ignores all new modifications. After committing cached journals, it enters an upgrade procedure that is shown from Figs. 5b, 5c, and 5d:

1) The elected standby accesses the global view and checks its own state. If it is in a junior state, it must stop upgrading and give up the lock. The re-election action will be performed at this time.
2) The elected standby modifies relevant states in the global view. It changes the state of previous active to standby or junior and sets itself to active. At this moment, operations from the previous active will be refused by all nodes.
3) New requests from clients and read service are allowed. Once server states are switched, new file operations may reach the elected standby. It receives and saves requests in memory but does not commit them until the upgrade process is finished.
4) To avoid missing operations (e.g., the previous active does not return success to all clients), the elected

standby flushes last cached journals to others in the replica group again. As the previous active may become standby and receive the same journals again, duplicated journals must be distinguished at this step. Each standby will decide whether to commit logs by comparing values of sn. Only if sn from the active is larger than the current maximum serial number, the standby applies journals and responds to it.

5) The elected standby receives register information from all servers in the replica group. It confirms and changes the state of each server in the global view. If a server does not have the same maximum sn, it is switched to junior. Otherwise the server will be assigned to standby or junior according to its previous state.
6) If more than one standby is registered successfully, the elected standby becomes the new active. It will launch the renewing process for junior at an appropriate time.
7) When a junior is upgraded to the standby state, the global view turns back to a stable status which is shown in Fig. 5d. The global view can also be stored in the SSP periodically for persistence. During the process of state transition, the elected standby will stop upgrading if any failures occur (e.g., standby fails or network failures happen to all others). The MAMS will launch the re-election process in the replica group at this time.

Benefiting from our namespace partitioning strategy [52], the client can reconnect to the new active directly and automatically after active-standby switching and resend requests when needed. As the process is completely transparent to applications, the file system does not see errors in case of failures.

4.2.3 Junior Renewing

Once the active has detected fatal errors, such as disk failures, it will be directly degraded to the junior state. The standby is often converted to the junior state when errors occur, i.e., node crashes or does not provide an ACK message to the execution of metadata synchronization. The junior is an intermediate state which cannot receive metadata from the active via the SSP and provide a hot standby. With the reduction of standbys, the file system will turn into an unreliable status. To avoid this risk, the MAMS adopts the renewing protocol to make the junior recover missing operations and become the standby.

During the runtime, the active scans the global view periodically and tries to launch the renewing process when there are juniors. It selects one server with the least gap in namespace state and creates a session for recovery at each time. In response, the junior receives commands and starts the task for upgrading. The active decides the renewing strategy according to the value of maximum sn from the junior. If the difference between them is large, the junior will first retrieve the namespace image or journal files. To check which journal files are missing, the junior needs to ask the active to retrieve the journal file list. As the namespace image and journal files are all stored in the SSP, the junior can obtain them from the SSP to reduce latency. Otherwise, if the junior cannot access the SSP, it will send request to the active to retrieve metadata files. All above operations are performed in the background and do not affect file system service. During the course of reading namespace image, the junior reconstructs the file and directory tree in memory and reaches a consistent state with the active gradually. As there is no sn associated with it (0 as default) and this phase can take a long time, the junior records the point that has been committed. It can continue to recover from other replicas in SSP at the last position and avoid retransmitting the whole files if there are any interruptions in the process. During the procedure of reading journal files, the junior records the current sn and sends it to the active periodically.

When there are few gaps in the values of sn, the active launches the final synchronization stage. Once the junior recovers all journals and returns the same sn, the active modifies the global view and changes its state to standby. Then the junior is upgraded, acts as standby node in the SSP again, and receives metadata from the active. It keeps an up-to-date namespace state in the form of hot standby. By renewing, more new backup nodes can also be added in the replica group at runtime. It significantly improves the reliability and availability of metadata service.

4.3 Global State Recovery Mechanism

While providing fault tolerance to keep the continuity of metadata service, there exists one important problem of global state inconsistency in case of multiple points of failure, i.e., more than one active servers fail. If excessive faulty servers occur or some servers cannot be recovered by the MAMS policy, the file system will still be in failure. At this time, it needs to restart all metadata servers and achieve failover for the entire file system. However, the status of each metadata server may not be in the same consistent state after restarting. The reason is two-fold. First, with the multiple metadata service, each metadata server manages a portion of the namespace and maintains part of the whole file and directory tree. Some operations, e.g., mkdir and delete, need to be performed as distributed transactions cross metadata servers for consistence guarantee. Since the distributed transaction corresponds to operations and journals in multiple metadata servers, it may cause journal inconsistent when failures occur. Second, although there are consistency protocols for distributed transaction coordination, it is difficult to reach a consensus between metadata servers after restarting as each server loads its own journals and is unaware of transaction states [28], [53].

To address this issue, we present a new approach based on the shared storage pool to achieve global state recovery. we use unified timestamp to make checkpoints [55] in distributed metadata servers, where each checkpoint represents a global consistent state at some point in time. In case of multiple failures, all metadata servers restart and load journals until reaching the same checkpoint, which ensures the file system recovers to a previous, consistent state.

The global state recovery mechanism is triggered for backup by external commands. To decide a timestamp for checkpoint, it selects one node from active servers as the time server so that other metadata servers send their system time to it. By comparing different system time, the time server selects an average value as the unified timestamp

and sends it back to other nodes. The negotiated timestamp may not match current system time with some metadata servers. It does not matter as the timestamp just indicates a time point for recovery. If all metadata servers receive the checkpoint timestamp, they begin to write unique flags in their journals with a collaborated way [56]. The time server first stops receiving client requests. Then it commits all metadata modifications in memory and writes a *barrier* flag at the end of journals. At last, the time server initiates requests to neighbor nodes, e.g., metadata servers in the same rack, for notification. The neighbor nodes receive requests and perform the same operations as the time server to make checkpoints. When all metadata servers write flags in journals, a global consistent checkpoint (unified system state) is stored in the SSP. Algorithm 3 describes the marking of a global consistent state in metadata servers.

Algorithm 3. Marking of a Global Consistent state in Metadata Servers

1 num = the number of active servers;
2 $MDS[num]$ is an array of active servers;
3 MDS_{time} = time server;
4 $neighbors_{mds}$ = neighbor nodes of mds;
5 T_{stamp} = unified time point;
6 $Flag$ = whether the barrier flag is marked in the journals;
7 **for** each $mds \in MDS[num]$;
8 **do**
9 **if** no MDS_{time} **then**
10 $i \Leftarrow rand\ number \bmod num$;
11 $MDS_{time} \Leftarrow MDS[i]$;
12 **end**
13 mds send system time to MDS_{time};
14 $T_{stamp} \Leftarrow T_{stamp} + system\ time\ of\ mds$
15 **end**
16 $T_{stamp} = T_{stamp}/num$;
17 MDS_{time} send T_{stamp} to $each\ mds\ in\ MDS[num]$;
18 MDS_{time} stop service and commit journals in memory;
19 $Flag \Leftarrow true\ in\ MDS_{time}\ at\ T_{stamp}$;
20 **for** each $neighbor\ node\ p \in MDS_{time}$;
21 **do**
22 MDS_{time} send a message mkr to p;
23 **if** $(!Flag)\ in\ p$;
24 **then**
25 $CLMark(p, neighbors_p)$;
26 **end**
27 **end**
28 void CLMark(node p, nodes $neighbors$) {
29 p receive mkr;
30 **if** $(!Flag)\ in\ p$; **then**
31 **then**
32 p stop service and commit journals in memory;
33 $Flag \Leftarrow true\ in\ p\ at\ T_{stamp}$;
34 **for** each $node\ q \in neighbors$;
35 **do**
36 $CLMark(q, neighbors_q)$;
37 **end**
38 **end**
39 }

When restarting all metadata servers for global state recovery, a timestamp needs to be set as recovered checkpoint. Then each active server reads its namespace images and merge metadata journals with them from the SSP to construct namespace structure. It stops loading metadata until meeting the *barrier* flag at the timestamp. The standby nodes also get the timestamp and reach the same state with the active when it registers at the active in each replica group. Then all metadata servers recover to a consistent checkpoint of file system status that is before failures. When the active restarts, it can write the namespace state in memory to a new image file and stores a snapshot in the SSP. The global *barrier* flag can be marked many times so that the whole file system can recover to different checkpoints. After global state recovery, the file system provides service and responds to clients again.

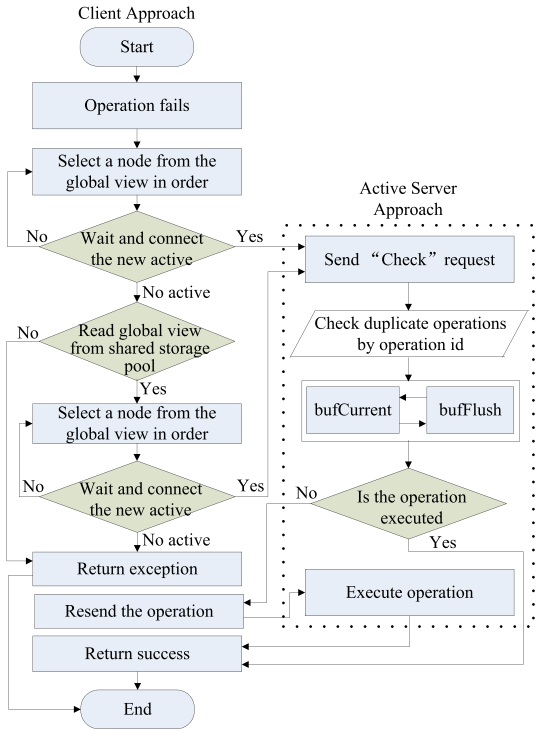


Fig. 6. Transparent Fault tolerance with smart-client.

4.4 Smart-Client Fault Tolerance Mechanism

To achieve transparent fault tolerance for upper applications [57], new smart-client is performed in our highly reliable policy. Unlike traditional manual service switching [4] or virtual IP approaches [14], the client can automatically connect the new active and cooperates with the MAMS policy for recovery. When failures occur, the last group of metadata operations may have been committed in the new active server before service switching. The client will resend requests for failed operations, and needs the new active server to distinguish duplicate requests for fault tolerance. Fig. 6 shows the flowchart of smart-client, in which the left part depicts client approach and the right dotted box depicts coordination of the active server.

The client first gets the global view from parameters or the configuration file. With hash-based methods for namespace partition [52], the client usually directly sends requests to the active in one replica group except for distributed transaction operations, such as *mkdir* (discussed in Section 5). It then waits and receives responses from the

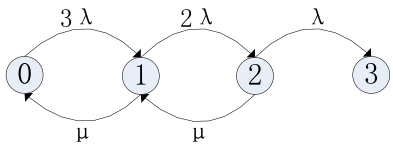


Fig. 7. Markov model of one active and two standbys in case of node failures.

active for getting the results. If there are any faults in metadata servers, the client gets errors of return results and starts the approach of fault tolerance, as listed in the steps below:

1. To find the new active more quickly, the client selects a node in the replica group from its global view for the last failed operation and attempts to connect it. The selected server may have no responses as the active election or state transition has not been finished. It does not matter as the client will connect the remaining nodes in the replica group orderly after several tries.
2. If a server is connected successfully, it means that the server is the new active and can provide metadata service. At this time, the client modifies its own global view and sends a "Check" message to the new active for detecting whether last failed operation has been executed. The message includes a unique operation id which is composed of client name and a monotonically increasing operation number. Different from (sn), it is only generated at the client side and already appended on each operation when the client sends it.
3. When a server provides metadata service, it uses double buffers to flush journals: *bufCurrent* for saving new incoming metadata and *bufFlush* for actual writing. The context of *bufCurrent* is switched to *bufFlush* if metadata operations are aggregated to a certain threshold. As *bufFlush* keeps the last group of modifications, the new active can parse it to obtain the operation id list that have been executed. The id list is like a hash set which saves multiple groups of metadata operations and can be updated with replacement strategy at periodic intervals. When receiving the "Check" message, the new active detects duplicate requests and judges whether to redo it.
4. If the operation has been executed, the new active tells the client to directly return success to applications. Otherwise, the client needs to resend the failed operation and waits the new active to execute it.
5. If the client cannot connect to all servers in the replica group after several times, it tries to read the global view from the shared storage pool. When the client obtains the latest view, it repeats from step 1.
6. When the client fails to visit the shared storage pool or find the new active after a certain number of tries, it no longer retries and returns errors to applications.

Combining the client with the active server for fault tolerance, the system ensures correctness of metadata service. As the client can quickly find the new active, it reduces the waiting time and avoids the effort of synchronizing status from the global view for client. With this approach, our highly reliable policy achieves a transparent fault tolerance when failures occur in the active-standby cluster.

5 Evaluation

In this section, we present the evaluation results of the proposed highly reliable metadata service. A file system prototype, CFS (Clover File System) [52], has been designed and implemented based on HDFS [4] with the reliable service for the validation and evaluation. It adopts multiple metadata servers to manage a global namespace and hash-based method for metadata partition. CFS is based on the SSP for metadata synchronization and replication and uses the highly reliable policies introduced earlier for metadata management. CFS also supports transaction operations across metadata servers, e.g., *mkdir*. For these operations, CFS maintains distributed operation consistency with consistency policy [52] for active servers across replica groups and applies active-standby synchronization in each replica group.

The experimental platform is a cluster with 20 nodes. Each node consists of four Intel Xeon X3320 Processors, 8-GB memory and one Gigabit network interface card, with Linux kernel 2.6.32. Each of them acts as the pool node in the SSP and stores image and journal files. For file system operations, multiple metadata modifications are aggregated before being submitted and written back to journals in an asynchronous way. To facilitate failure detection and auto service switch, the Zookeeper [47] was used to monitor nodes, trigger events and maintain the global view for active-standby server states.

5.1 Reliability Analysis

We first analyze the reliability of the active-standby cluster in theory. We use the Markov Model [58], [59] to evaluate the system reliability when adding backup nodes. In our design of multiple metadata servers, each active server and its backups form the replica group which performs service switching for managed namespace partition. As the cluster reliability can be described as an exponential distribution, the reliability of each replica group can be computed as below:

$$R(t) = e^{-t/MTTF}, \tag{1}$$

where the $MTTF$ is the mean time to failure of replica group. Supposing the system meets following conditions: (1) the failure probability of each node is independent and follows exponential distribution which is set to λ, i.e., $MTTF_{server} = 1/\lambda$; (2) the fault recovery rate of each node is set to μ, that is $MTTR_{server} = 1/\mu$ in which $MTTR$ is the mean time to recovery. Taking the case of one active and two standbys in one replica group, we build the Markov model for the system failure cases. As Fig. 7 shows, state 0 represents normal work status and state 1 represents one node failure. State 2 means two node failures and the replica group can only provide read service. State 3 represents all node failures of the replica group in which the file system can no longer provide metadata service.

At time t, the probability of system status being from 0 to 3 are $P_0(t)$, $P_1(t)$, $P_2(t)$ and $P_3(t)$, respectively. The initial value meets the following condition:

$$P_0 = 1, P_1(0) = P_2(0) = P_3(0) = 0. \tag{2}$$

According to Markov model in Fig. 7, differential equations can be established as follows. The flow-in and flow-out relationship is clearly seen in these equations. For

TABLE 1
Reliability Improvement by Adding Backup Nodes

Improvement Ratio	Number of Added Backup Nodes for One Active				
	1	2	3	4	5
$MTTF_n/MTTF_1$	10^3	10^6	10^9	10^{12}	10^{15}

instance, the total flow out of state 0, $P_0'(t)$, is given by $-3\lambda P_0(t) + \mu P_1(t)$ as there is $3\lambda P_0(t)$ out of state 0 and $\mu P_1(t)$ into state 0 [60].

$$P_0'(t) = -3\lambda P_0(t) + \mu P_1(t) \tag{3}$$

$$P_1'(t) = 3\lambda P_0(t) - (2\lambda + \mu)P_1(t) + \mu P_2(2) \tag{4}$$

$$P_2'(t) = 2\lambda P_1(t) - (\lambda + \mu)P_2(t) \tag{5}$$

$$P_3'(t) = \lambda P_2(t). \tag{6}$$

From above equations, the $MTTF$ can be computed for the replica group with one active and two standbys in case of failures:

$$MTTF = \int_0^\infty (P_0(t) + P_1(t) + P_2(t))\, dt. \tag{7}$$

Take each initial value into the equation and we have the following results:

$$MTTF = \frac{\mu^2 + 4\lambda\mu + 11\lambda^2}{6\lambda^3}. \tag{8}$$

This is the mean time to failure of replica group in MAMS policy. If only one server provides metadata service without standbys, the value of $MTTF_1$ is $1/\lambda$. Thus we have:

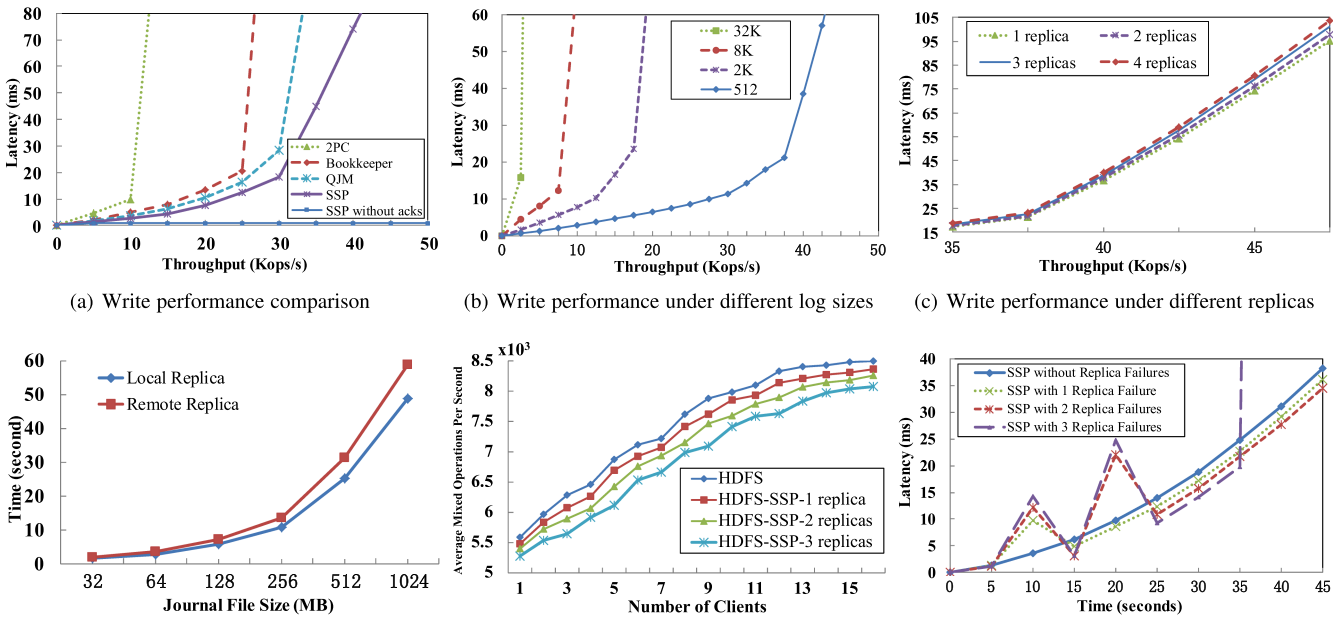
$$\frac{MTTF}{MTTF_1} = \frac{\mu^2 + 4\lambda\mu + 11\lambda^2}{6\lambda^2}. \tag{9}$$

As in actual applications the value of μ is considerably larger than λ, the result of the above equation can be estimated. For instance, $MTTF/MTTF1 \approx 135601$ in the case of $\mu = 0.9, \lambda = 0.001$. It means that the system reliability is improved by 6 magnitudes comparing two standbys with a single metadata server. The availability of the file system will be further improved if more standbys are added in the replica group. With the above analysis, Table 1 reports the reliability improvements when adding different number of standbys, where $MTTF_n$ means the MTTF by adding n additional backup nodes.

5.2 Evaluation of Shared Storage Pool

In a highly reliable metadata service, the metadata server records logs for metadata persistence when it responds to client requests. The backup node also synchronizes metadata modifications with the active server for standby. In our design, we use the SSP for distributed metadata storage and server state synchronization. To compare SSP with other approaches, typical policies, including the 2PC [25], Hadoop Quorum Journal Manager (QJM) [7], and BookKeeper log system [31], were implemented or deployed on multiple nodes for metadata synchronization and replication. The Hadoop QJM provides a feature to share edit logs between the active and standby metadata servers in HDFS, where the JournalNodes (metadata replication nodes) are set to default value 3. BookKeeper is a system to reliably log streams of records. The tests first compared the write performance of different policies by using one node as the log writer. Then it used SSP as the metadata storage for namenode in HDFS to evaluate metadata operation performance.

Fig. 8 shows the performance of journal storage in SSP. Fig. 8a compares the journal write performance under different policies with 3 replicas. The x-axis represents write counts per second and the y-axis represents the average delay latency. All logs were flushed asynchronously with



(a) Write performance comparison (b) Write performance under different log sizes (c) Write performance under different replicas

(d) Read performance with different journal sizes in SSP (e) Metadata operation peformance compared with HDFS (f) Journal write performance with different replica errors

Fig. 8. Performance of journal storage in SSP.

each 1 KB size. From the results, it can be observed that the performance of 2PC was the worst among all policies. Its latency was nearly 2, 3, and 4 times of the values of the BookKeeper, QJM, and SSP, respectively. This is because the journal replication in the 2PC needs four round trip messages, which considerably increases the waiting time and load of nodes. BookKeeper is a logging storage system which writes replicas to assigned redundant nodes simultaneously. However, in order to ensure strong consistence, BookKeeper waits for an operation to be completed until all replicas have been written successfully. Otherwise, it selects other nodes to rewrite journals via a round-robin way when replica errors occur. The QJM is a log sharing storage used between the active and standby namenodes. It can tolerate at most $(N-1)/2$ failures but needs additional overhead for journal synchronization and lacks the mechanism of fault tolerance. Compared with the BookKeeper and QJM, SSP shows a better result, which improved the performance by 40.51 and 23.46 percent, respectively when writing 25K logs per second. This is because SSP adopts a replication policy that combines local files and remote copies. It reduces the communication overhead in which SSP returns success only if more than half replicas are written. All write latency increased gradually with the increase of the throughput except in the case of SSP without acks (which means writing journal copies without waiting for replication node response in SSP). It is due to the processing capability limit of the writer node.

Fig. 8b reports the write performance under different log sizes with 3 replicas in SSP. It can be seen that the SSP achieved better performance with less size logs. For example, the average write latency has been increased nearly 20 times when the log size varied from 512 B to 32 KB. In distributed file systems, journal storage belongs to a small chunk of writing data operations. It verified the effectiveness of SSP for metadata synchronization and replication. With the appropriate log size 512 B, Fig. 8c gives the write performance under different replicas in SSP. The results show that the write latency is increased by 2.81, 2.62 and 2.95 percent when adding one replica. It improves metadata redundancy with negligible overhead on write performance.

As log records are stored in the journal file with multiple replicas in SSP, we performed tests to measure the read performance by using the namenode in HDFS to read journals. Fig. 8d shows the time spent for reading journals with different sizes and replicas. The results reveal that the read performance of local journal for namenode was about 22.32 MB/s, which is similar with that of HDFS. The time increased with the increase of journal sizes. As the namenode would replay logs to construct namespace in memory, it spent additional overhead for reading journals. When the namenode read logs from remote replicas, there is a performance degradation which was reduced by 19.59 percent on average compared with local copies. Network communication caused delays, but the effect is negligible. The SSP still achieved impressive read performance with the replication strategy of increasing metadata availability.

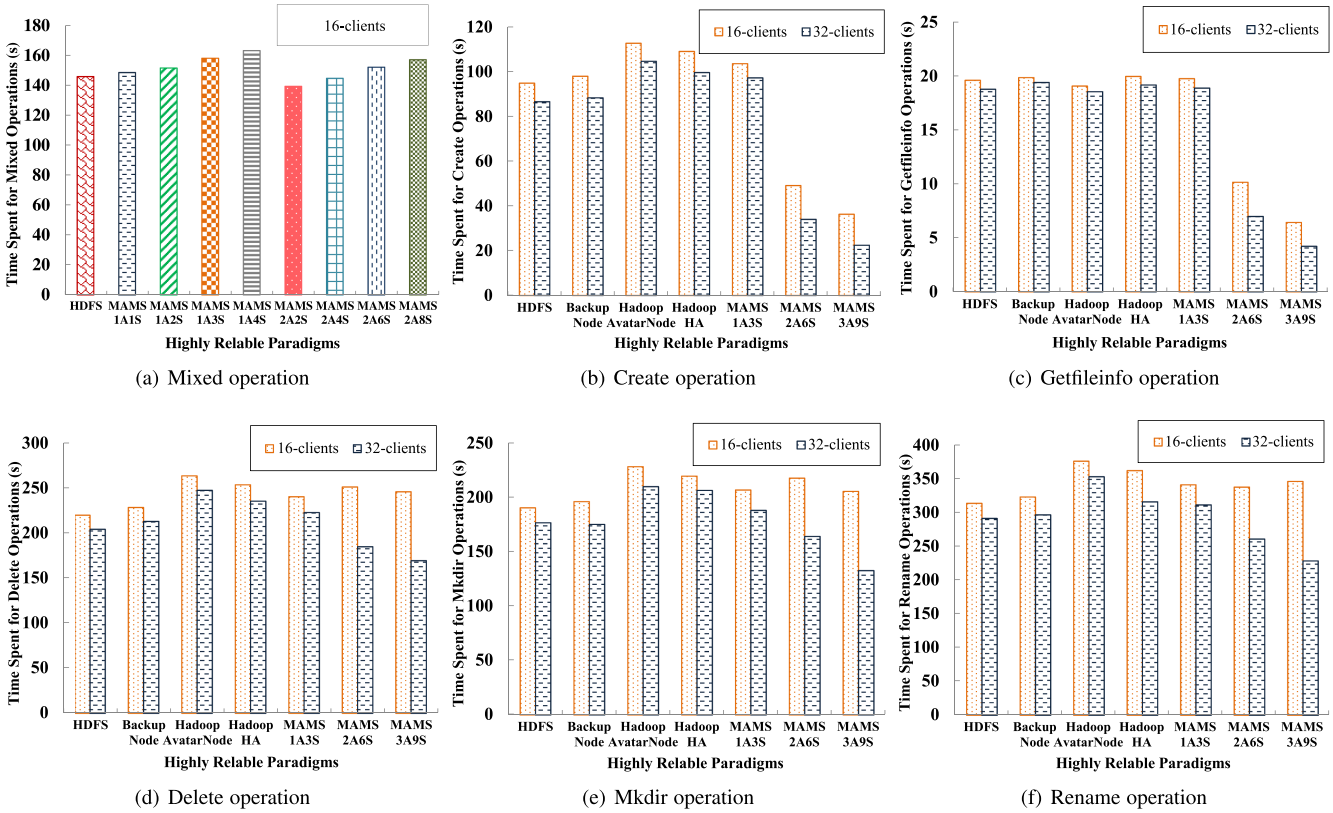
We have conducted another series of tests to measure the performance influence on metadata operations with the journal write policy in SSP. These tests used the namenode to provide metadata service and multiple clients on different nodes for payload. Fig. 8e shows the metadata operation performance comparison by using SSP as journal storage in HDFS. It can be observed that metadata operations with one local journal achieved the best performance, which is nearly equal to that of HDFS. With the increase in replica number, the average operations per second were reduced. This is because the namenode would replicate journals to pool nodes through the SSP for multiple copies, which incurs additional data transmission overhead. The average performance decreased by less than 6.1 percent when adding two additional remote replicas, namely HDFS-SSP-3 replicas. The results confirm that the SSP achieved metadata synchronization and replication with little effect on the performance.

To verify metadata restoration in the SSP, we test journal write performance in case of failures. Fig. 8f depicts the journal writing performance in case of different replica errors. Among them, the vertical axis is average access latency. In these tests, we increase the workload gradually (accordingly the SSP latency without replica failures increases for its curve) and generate replica errors by killing pool node processes. When there was one replica error at the 10th seconds, the latency increases as much as three times for failure free cases. This is because the SSP waits for the response from the failed pool node and removes it when timeout, which incurs an additional overhead. But the SSP was recovered to normal performance subsequently. With our journal replication approach, the SSP returns success when more than half numbers of replicas are written. Although the written replicas are decreased, the performance of SSP is improved with the reduction of response time and waiting latency. The experiments show the same results when there were two replica faults. In the case of more than three replica errors, the journal writing would fail because the number of written replicas was less than half. It resulted in the linear increase of the average latency.

In conclusion, the experiments show that the SSP achieved superior write performance and short recovery time (from 5 to 15 milliseconds) even in the case of replica failures. These tests have confirmed the effectiveness and reliability of our metadata synchronization and replication strategy.

5.3 Evaluation of MAMS for Performance Overhead in Case of Failures Free

Based on the SSP, the MAMS policy achieves metadata synchronization and replication among the active server and standby nodes. It provides a highly reliable mechanism for metadata service, though it may have an impact on normal metadata operations. To measure the overhead on normal metadata operations, the experiments were conducted under failure free cases. As the CFS is an implementation of multiple metadata service based on HDFS, we first compare it with vanilla HDFS by configuring HDFS with different active and standby nodes. Then we compare MAMS with typical highly reliable systems, including BackupNode [4], Hadoop AvatarNode [14], and Hadoop HA [7], to observe the metadata operation performance. All three comparison systems are based on HDFS and use the primary-backup policy (one active and one standby server) for metadata reliability. For Hadoop HA, it uses the QJM mechanism for metadata sharing in which the JournalNodes (metadata replication nodes) are set to default value of 3.



(a) Mixed operation (b) Create operation (c) Getfileinfo operation

(d) Delete operation (e) Mkdir operation (f) Rename operation

Fig. 9. Metadata operation performance compared with other reliability mechanisms.

The experiment results are shown in Fig. 9. Each test was performed with total 1 million operations with multiple clients (16 or 32 clients) and metadata operations. Fig. 9a shows the mixed operations (including create, getfileinfo, delete, etc.) performance of HDFS and CFS with the MAMS policy. The CFS was configured with different active and standby servers, e.g., MAMS-2A8S means 2 actives and 8 standbys in 2 replica groups. The HDFS adopts a single metadata server without any reliable mechanism. The experiment results reveal that the metadata operation performance was reduced in CFS from MAMS-1A1S to MAMS-1A4S compared to HDFS. Note that the performance of CFS was gradually decreased when adding standbys in each replica group (i.e 1A1S achieves higher performance than 2A2S). The reason is that the active server needs to synchronize journals to more standbys, which will increase additional time. The performance of metadata operation with 1A4S was declined with 3.87, 4.06 and 3.25 percent respectively by adding one standby for the active metadata server. The MAMS policy, however, was built upon the SSP which reduced the synchronization overhead and was not a performance bottleneck.

For more detailed evaluations, we compare MAMS with other highly reliable policies by performing different metadata operations. Figs. 9b, 9c, 9d, 9e, and 9f shows the performance results of measurements. The tests used multiple clients on different nodes to provide the workload with each performing the same number of operations. The experiment results reveal that, except the read-only operation *getfileinfo*, the metadata operation performance was reduced with the primary-backup mechanisms and MAMS-1A3S compared to HDFS. This is because they need real-time state synchronization between the metadata server and backup nodes, which results in additional overhead. Compared with others, the BackupNode incurred less time but it does not guarantee metadata consistency. For MAMS-1A3S, the performance was higher than the Hadoop AvatarNode and Hadoop HA even using multiple standbys. This is mainly due to two aspects: one is the journal synchronization strategy with the SSP which greatly reduced the additional overhead; the other is the MAMS policy can effectively perform service switching and achieve failover.

From Figs. 9b and 9c, it can be observed that the performance of CFS with MAMS-2A6S and MAMS-3A9S is higher than HDFS and the primary-backup mechanisms for *create* and *getfileinfo* operations. This is due to the metadata partition strategy in CFS in which the metadata servers can perform metadata operations simultaneously. The performance of CFS is further improved when the client number varies from 16 to 32. For example, the CFS with MAMS-3A9S only spends 26.98 and 22.47 percent time of HDFS for *create* and *getfileinfo* operations at 32-clients. The effect of adding clients is not significant for systems with HDFS and the primary-backup mechanisms as they use one metadata server which reaches the performance limitation at 16-clients. It proves that the CFS achieves good performance for the two operations even if using the MAMS policy for reliability.

The other three types of operations, including *delete*, *mkdir* and *rename*, belong to distributed transactions in the CFS. Besides state synchronization, the transaction consistency cross metadata servers will have an impact on these operations. As shown in Figs. 9d, 9e, and 9f, it needs more time spent for operations in CFS. The performance is improved when adding client load, e.g., the CFS with MAMS-2A6S and

TABLE 2
MTTR of Different Reliable Metadata Management Systems

Image (MB)	MTTR (s)			
	MAMS-1A3S	BackupNode	Hadoop Avatar	Hadoop HA
16	5.893	2.784	27.362	15.351
32	6.376	5.326	31.574	17.439
64	6.531	9.653	30.721	18.624
128	5.742	22.928	29.273	16.372
256	5.436	36.431	32.805	19.016
512	6.795	78.365	31.446	17.853
1024	6.081	142.513	33.239	19.193

MAMS-3A9S achieves better performance than HDFS and the primary-backup mechanisms at 32-clients. The MAMS policy may affect some operation performance of the CFS but significantly enhances the metadata service reliability (in general, any reliable metadata management strategy sacrifices the performance for reliability). It is worthy to trade slight performance reduction for the entire file system reliability in return.

In summary, the experiment results indicate that the MAMS policy can significantly improve metadata service reliability and keep server states consistent in the same replica group with little effect on the performance.

5.4 Evaluation of MAMS for Failover Time in Case of Failures

This series of tests measured the failover time of different reliable metadata management systems discussed in the above subsection. They are all implemented based on the HDFS and increase the metadata reliability guarantee compared to the vanilla HDFS. We also observed and analyzed the time needed for different stages in the MAMS in these tests, which includes active election, active-standby switching and client reconnection. The CFS was configured with 1A3S and other systems adopt the primary-backup strategy. In the Hadoop HA, the number of JournalNodes was set to 4. For fault detection, the heart beat interval and session timeout of ZooKeeper were set to 2 and 5 seconds, respectively. Metadata server failures were generated by shutting down processes, unplugging the network cables, etc.

During the experiments, client interfaces were called to access the file system continually and the failover time was measured. We choose the mean time to recovery (MTTR) as the metric for comparison. It can be computed by subtracting the timestamp of operation failure and that of success when metadata service is unavailable and recovered, respectively. Each test was performed 10 times, and the average MTTR was computed as below:

$$MTTR = \frac{\sum_{k=1}^{times}\left(Time_{return\ failure} - Time_{return\ success}\right)}{Times}.$$

Table 2 reports the MTTR of different systems. The first column lists the image size, which is the file size of a namespace image that saves the entire file and directory tree structure of the distributed file system. The image size can indicate the file system scale, e.g., there are more than 7 million files when it is about 1 GB. The tests were conducted using *create* operations to create files. We can observe that the MTTR of BackupNode increased gradually with file system scale expansion from 2.784 to 142.513 seconds. This is mainly because its backup node needs to recollect block locations before taking the place of the primary. Compared with the BackupNode, Hadoop Avatar, and Hadoop HA, the average failover time of the MAMS are 14.35, 19.77, and 34.54 of them respectively. The results verified two aspects. One is that the overhead of failover in MAMS is relatively low. By exclusive of session timeout (default 5 seconds), the time spent for active election and active-standby switching are less than 100 ms and 350 ms, respectively. The other is that clients can quickly connect to the new active and achieve transparent fault tolerance.

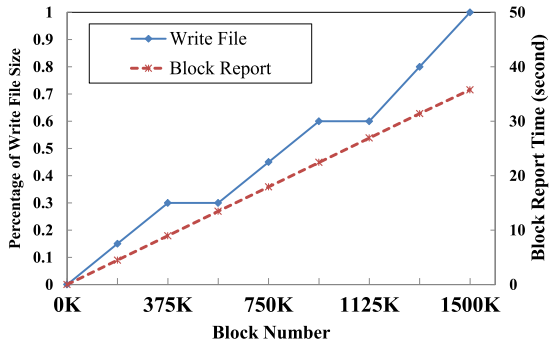


Fig. 10. Failover ability of file writing. The Write File line is plotted base on the left axis and the Block Report line is plotted based on the right axis.

5.5 Evaluation of MAMS for Recovery Efficiency in case of Failures

In the above section, we have measured the failover time comparing MAMS with other highly reliable paradigms. The results are all tested based on a single point of server failure. This subsection continues to verify the efficiency of the MAMS for recovery at different failure scenarios. The CFS is configured with one replica group including one active and three standbys (1A3S). Metadata server failures are generated by shutting down processes, taking out/plugging back network wires and etc. The tests are performed from three aspects: file writing operations, metadata operations and junior renewing.

5.5.1 File Write Recovery

Like HDFS, the CFS adopts the method of decoupling metadata and data operations. It splits the file into blocks and distributes them among datanodes with multiple replicas. To utilize memory space effectively, the metadata server only maintains mapping relation between files and blocks. Actual block locations are reported by datanodes when the system starts up. Though MAMS provides a highly reliable policy for metadata service, the file I/O is also important for applications [61]. Besides structures of files and directories in memory, the state recovery includes block information collection for file locations. To evaluate file I/O recovery, the test writes a total 100 GB file with the block being set 64 KB. The CFS is configured with 1A3S with 3 replicas.

Fig. 10 shows the failover ability of file writing in CFS. The horizontal axis represents the number of reporting file blocks, the left vertical axis indicates the proportion of file

TABLE 3
Test scenarios and Server State Transition

State	Test A				Test B				Test C			
	MDS	BN	BN	BN	MDS	BN	BN	BN	MDS	BN	BN	BN
1	A	S	S	S	A	S	S	S	A	S	S	S
2	–	S	S	S	A	S	–	–	–	S	S	S
3	–	A	S	S	A	S	J	J	–	A	S	S
4	S	A	S	S	A	S	S	S	J	A	S	S
5	S	–	S	S	–	–	S	S	S	A	S	S
6	S	–	A	S	–	–	A	S	S	–	–	S
7	S	S	A	S	J	–	A	S	S	–	–	A
8	S	S	–	S	S	–	A	S	S	J	–	A
9	S	S	–	A	S	J	A	S	S	S	J	A
10	S	S	S	A	S	S	A	S	S	S	S	A

written and the right one is block reporting time. When writing the file, we generate metadata server errors twice to observe the results. As seen in the figure, the writing operations were suspended because the active failed when writing 30 and 60 percent of the file size. In CFS, the client mainly communicates with datanodes and transmits file blocks to them directly after creating files. As blocks are allocated by the metadata server, the client can only continue to write the file until it waits for the new active to provide service in case of failures. During service switching, the MAMS policy may cause temporary data transmission interruption but it will recover the file writing after failover. For a hot standby, the datanodes send block information to both the active and standbys periodically. Due to time lag, the new active may not receive latest block locations and can not access needed data. The problem can be analyzed from two aspects. One is to reduce the time interval of block reporting but may result in lots of network communication overhead in a large scale storage system. The other is to use the heart beat messages to perform block querying and updating. With our policy, the results indicate that the time is less than 40 seconds when datanodes send about 1.5 million blocks to all active and standby servers. So there is little influence on data access correctness. When reading a file, the client can get the file block locations directly from the new active when errors occur. The results prove that our highly reliable policy can also achieve fault tolerance for file I/O.

5.5.2 Metadata Recovery

This section reports the evaluation results of the metadata recovery ability of MAMS in various scenarios. We used multiple client processes on different nodes to provide overload. The tests include continuous *create* and regular *mkdir* operations. Files are distributed among multiple directories with average requests per second being counted. Table 3 reports different test scenarios and corresponding server state transition. The states of nodes include active (denoted as A), standby (denoted as S) and junior (denoted as J). The symbol "-" means the server is still in a fault state.

Three groups of tests were performed in our experiments. Errors were generated through modifying the global view to make the active lose the lock (Test A), unplugging and reconnecting network wires (Test B), and restarting processes (Test C). Fig. 11 shows the failover ability of metadata operations in the CFS. The vertical axis is average requests per second and the horizontal axis is the test time.

As shown in Fig. 11a, the active lost the lock at the moment of 60 seconds. It triggered an active election and state transition in response to state 2 in Table 3. At this point, the average time of metadata operations was not reduced to 0 immediately. It is because the active has returned partial success results to clients. Subsequently, the active stopped the service and performed a fault-tolerant processing from 62 to 68 seconds. The file system did not respond to clients anymore. After service switching, the state of each node was transformed to state 4. The original active registered to the new one as a standby. When the new active began to provide service, clients may resend requests for incorrect results. Therefore, the curve has a slight upward trend at the time of 70 seconds. The duplicated message handling in the MAMS will avoid the problem of incorrect metadata operations. At last, the file system restored to the normal performance before failures in which the new active continued to provide metadata service. The same operations were also performed at the time of 120 and 180 seconds. Experiment results indicate that the MAMS policy can elect the new active and achieve server takeover quickly to keep the continuity of file system service.

Test B and test C show similar results when generating errors by taking out/plugging back network wires and shutting down/restarting server processes for several seconds. As expected, the MAMS policy can perform active-standby switching, achieve upgrade for juniors by the renewing protocol, and tolerate various failures for metadata servers. Though there is some performance degradation, it still significantly improves the reliability of file system with a fast recovery. These experiment results verify the efficiency of the MAMS policy well.

5.5.3 Junior Renewing

The MAMS policy maintains automatic state transition among nodes in the replica group for fault tolerance. This test further verifies the failover ability of renewing protocol for junior nodes or new added backup nodes. The experiment

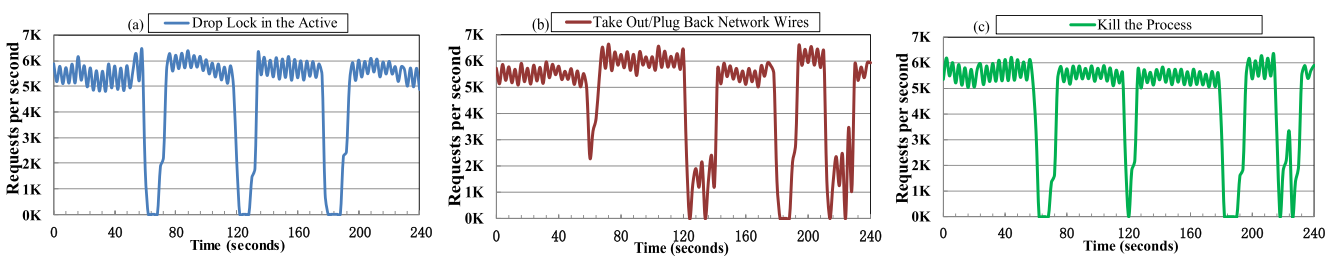


Fig. 11. Failover ability of metadata operations.

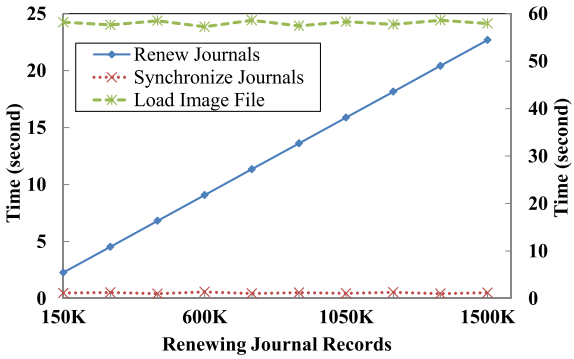


Fig. 12. Time spent for junior renewing.

results are as shown in Fig. 12. The horizontal axis is the number of log records for renewing journals. The left vertical axis is the time required to renew or synchronize journals and the right one is the time for loading image. When upgrading, the junior will first load namespace from the image file if it misses lots of operations compared with the active. The top curve represents the time when the junior reads fixed image in which the initial size is 1 GB. After loading image, the junior reads remaining operations from shared journals in SSP according current submitted sn number. It can be seen that time of renewing journal increases with the growth of log records. If there are few gaps in namespace state, the junior can synchronize latest operations from the active directly. Experiment results show that the main cost for junior upgrading is for reading the image and journal files from the SSP. As the procedure is achieved in the background, the active can still provide service at this time. Only when the active performs final synchronization (as shown in the bottom curve), it stops to receive client requests for journal transmission. It spent little time in which the value is about 22.78 milliseconds when synchronizing 10,000 log records. So there is negligible effects on normal file system operations. When new backup nodes are added in the active-standby cluster, it can be upgraded from the junior state to standby state by the same way. Experiment results verify the effectiveness of renewing protocol for service fault tolerance. It will significantly improve the file system availability when more backup nodes are added in the active-standby cluster.

5.6 Evaluation of Global State Recovery Mechanism

In this section, we evaluate the global state recovery and snapshot with our highly reliable metadata service. Each test loads or writes metadata files with a total 5 GB size in which the unified timestamp is pre-written in the journals.

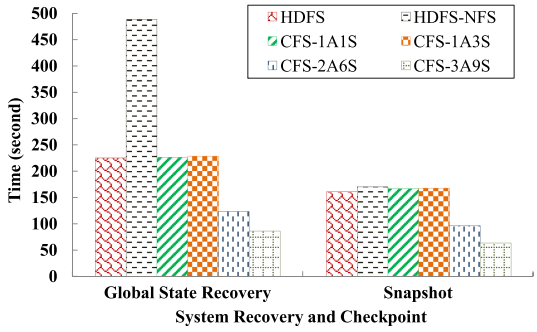


Fig. 13. Time spent for global state recovery.

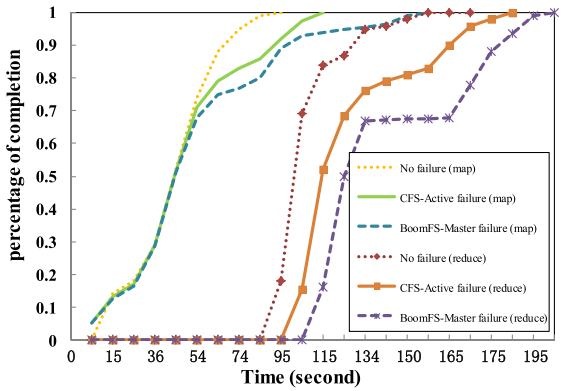


Fig. 14. Run time comparison for Hadoop programs.

For comparison, we configured NFS [30] as a shared storage for HDFS to store metadata files, namely HDFS-NFS. As shown in Fig. 13, the time spent for global state recovery is approximately equal for HDFS, CFS-1A1S and CFS-1A3S. It is because the CFS can load the metadata information from local files in the SSP directly, which achieves the similar performance with HDFS. For HDFS-NFS, the cost is nearly two times more than that of HDFS. The network communication of accessing metadata files in NFS results in additional overhead. For CFS-2A6S and CFS-3A9S, it spent less time for recovery than HDFS as multiple metadata servers partitioned the global namespace and loaded the image simultaneously. Besides, the backup nodes can reach the same metadata state with the active parallelly. Benefited from our global state recovery mechanism, the metadata servers in CFS can effectively reach to the same timestamp and recover to a consistent state after starting up.

The snapshot is an image file for saving the namespace state in memory. It is made by the namenode in HDFS or the active server in CFS. From the figure, it can be seen that CFS-1A1S and CFS-1A3S spent the similar time with HDFS and HDFS-NFS. As backup nodes need not make the snapshot, the CFS can quickly write a new image file in the SSP. For HDFS-NFS, there is no significant performance degradation to flush the image on NFS as the write cache reduces the latency. When using multiple metadata servers, such as CFS-2A6S and CFS-3A9S, the CFS spent less time for making snapshot. This is because the metadata servers partition the namespace in which each one only need to flush part of the whole file and directory tree. The experiments prove that our highly reliable metadata service can achieve efficient global state recovery for file system.

5.7 Evaluation of Highly Reliable Metadata Service for Hadoop Applications

In this section, we evaluate the tolerance ability of our highly reliable metadata service with standard MapReduce applications as use cases. The CFS file system provides compatible interfaces with the HDFS and supports mass data processing in the MapReduce framework. To verify transparent failover for upper applications, we compared the CFS with the BoomFS, another typical reliable system with multiple metadata servers based on the HDFS. The test generated a metadata server error for measuring the program completion time with the same environments. It runs a Hadoop wordcount job on a 5 GB input file. The CFS was configured with 3A9S.

Fig. 14 shows the cumulative distribution of the percentage of completed MapReduce jobs over time in case of failures. The data of Boom-FS including normal and failure operations comes from [23]. As shown in the figure, it had an influence on the program when the metadata server failed. Compared with the Boom-FS, the CFS exhibited better failover performance when errors occurred. Its completion time of map and reduce jobs was less than the Boom-FS, by 28.13 and 9.76 percent respectively. The reduce jobs of the Boom-FS have a phenomenon of suspension. As it took time to finish map jobs for recovery, the reduce jobs needed the former to write intermediate results into the file system before continuing subsequent operations. It added additional waiting time. On the contrary, the fault tolerant mechanism in the MAMS ensured fast taking over for metadata service. It had little effect on the MapReduce execution. The experiment results indicate that our highly reliable policy can achieve transparent failover for upper applications and ensured expected behavior in case of failures.

6 Conclusion

Metadata reliability is critical to large-scale storage systems as it will affect file system operations and application performance. In this paper, we propose a novel highly reliable metadata service that fuses multiple new methodologies to improve distributed file system reliability and availability. We have introduced a new design of an active-standby architecture to manage replica groups for multiple metadata servers and adopted four collaborative and tightly connected modules to achieve fault tolerance in case of failures. We have implemented the highly reliable metadata service in Clover file system (CFS), a large-scale distributed file system for mass data processing compatible with the Hadoop platform. Note that these methods can also be used in other parallel/distributed file systems for metadata service reliability. Experiment results confirm that this new reliable metadata service can significantly improve the reliability of file systems with negligible influence on metadata operations and ensure fast, transparent failover. In future work, we will explore the integration of data availability with fault tolerance mechanisms and consolidation techniques to further improve file system reliability.

Acknowledgments

This research is supported by the Beijing Municipal Science and Technology Project under grant Z191100007119002. This research is supported in part by the National Science Foundation under grant CNS-1338078, CNS-1362134, CCF-1409946, CCF-1718336, OAC-1835892, and CNS-1817094. This research is also supported in part by the National Science Foundation of China No. 61572377, the Natural Science Foundation of Hubei Province of China No.2017CFC889, and the Fundamental Research Funds for the Central Universities No. 2018QNA5015.

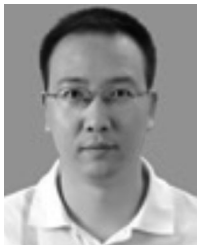
References

[1] P. J. Braam, "The Lustre storage architecture," White Paper, Cluster File System, Inc., Oct. 2003.
[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proc. 4th Annu. Linux Showcase Conf.*, 2000, pp. 391–430.
[3] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *Proc. ACM Symp. Oper. Syst. Principles*, 2003, pp. 29–43.
[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. Int. Conf. Massive Storage Syst. Technol.*, 2010, pp. 1–10.
[5] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.
[6] S. Sinnamohideen, R. Sambasivan, J. Hendricks, K. Liu, and G. Ganger, "A transparently-scalable metadata service for the Ursa Minor storage system," in *Proc. USENIX Annu. Techn. Conf.*, 2010, p. 13.
[7] "HDFS federation," 2017. [Online]. Available: https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/Federation.html
[8] A. Roy, H. Zeng, J. Bagga, and A. Snoeren, "Passive realtime datacenter fault detection and localization," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2017, pp. 595–612.
[9] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *J. Supercomput.*, vol. 65, no. 3, pp. 1302–1326, 2013.
[10] D. Fiala, F. Mueller, C. Engelmann, K. Ferreira, R. Brightwell, and R. Riesen, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. 78.
[11] X. Tang, J. Zhai, B. Yu, W. Chen, W. Zheng, and K. Li, "An efficient in-memory checkpoint method and its practice on fault-tolerant HPL," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 758–771, Apr. 2018.
[12] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. Supinski, N. Maruyama, and S. Matsuoka, "FMI: Fault tolerant messaging interface for fast and transparent recovery," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2014, pp. 1225–1234.
[13] R. Garg, T. Patel, G. Cooperman, and D. Tiwari, "Shiraz: Exploiting system reliability and application resilience characteristics to improve large scale system throughput," in *Proc. 8th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2018, pp. 83–94.
[14] D. Borthakur, et al., "Apache Hadoop goes realtime at Facebook," in *Proc. SIGMOD Conf.*, 2011, pp. 1071–1080.
[15] C. Min, S. Lee, and Y. Eom, "Design and implementation of a log-structured file system for flash-based solid state drives," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2215–2227, Sep. 2014.
[16] J. Yoo, J. Oh, S. Lee, Y. Won, J. Ha, J. Lee, and J. Shim, "OrcFS: Orchestrated file system for flash storage," *ACM Trans. Storage*, vol. 14, no. 2, 2018, Art. no. 17.
[17] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surveys.*, vol. 15, no. 4, pp. 287–317, 1983.
[18] S. C. Tweedie, "Journaling the Linux ext2fs file system," in *Proc. 4th Annu. LinuxExpo*, 1998.
[19] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, Art. no. 19.
[20] Z. Zhang, et al., "A light-weight log-based hybrid storage system," *J. Parallel Distrib. Comput.*, vol. 118, no. 2, pp. 307–315, 2018.
[21] A. Oriani and I. C. Garcia, "From backup to hot standby: High availability for HDFS," in *Proc. Symp. Reliable Distrib. Syst.*, 2012, pp. 131–140.
[22] F. Wang, et al., "Hadoop high availability through metadata replication," in *Proc. 1st Int. Workshop Cloud Data Manage.*, 2009, pp. 37–44.
[23] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears, "Boom: Data-centric programming in the datacenter," Tech. Rep. UCB/EECS-2009–113, 2009.
[24] T. Hsu and A. Kshemkalyani, "Value the recent past: Approximate causal consistency for partially replicated systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 212–225, Jan. 2018.
[25] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Berlin, Germany: Springer, 2011.
[26] "Three-phase commit protocol," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Three-phase_commit_protocol
[27] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. Ports, "Building consistent transactions with inconsistent replication," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 263–278.
[28] J. Xiong, Y. Hu, G. Li, R. Tang, and Z. Fan, "Metadata distribution and consistency techniques for large-scale cluster file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 5, pp. 803–816, May 2011.
[29] A. Hatzieleftheriou and S. V. Anastasiadis, "Host-side filesystem journaling for durable shared storage," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 59–66.

[30] R. Sandberg, et al., "Design and implementation of the Sun network filesystem," in *Proc. Summer USENIX Conf.*, 1985, pp. 119–130.
[31] "Apache software foundation," 2017. [Online]. Available: http://zookeeper.apache.org/bookkeeper
[32] F. Chang, et al., "Bigtable: A distributed storage system for structured data," in *Proc. USENIX Oper. Syst. Des. Implementation*, 2006, p. 15.
[33] J. Corbett, et al., "Spanner: Google's globally-distributed database," in *Proc. USENIX Oper. Syst. Des. Implementation*, 2012, pp. 251–264.
[34] Y. Lin and H. Shen, "EAFR: An energy-efficient adaptive file replication system in data-intensive clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1017–1030, Apr. 2017.
[35] S. He and X. Sun, "A cost-effective distribution-aware data replication scheme for parallel I/O systems," *IEEE Trans. Comput.*, vol. 67, no. 10, pp. 1374–1387, Oct. 2018.
[36] J. Zhou, Y. Chen, and W. Wang, "Attributed consistent hashing for heterogeneous storage systems," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, 2018, Art. no. 23.
[37] R. Shi and Y. Wang, "Cheap and available state machine replication," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 265–279.
[38] T. Kobus, M. Kokociński, and P. Wojciechowski, "Hybrid transactional replication: State-machine and deferred-update replication combined," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1499–1514, Jul. 2018.
[39] D. Fetterly, et al., "TidyFS: A simple and small distributed file system," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, p. 34.
[40] M. Isard, "Autopilot: Automatic data center management," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 60–67, 2007.
[41] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.
[42] B. Liskov, et al., "Replication in the Harp file system," in *Proc. Symp. Operating Syst. Principles Conf.*, 1991, pp. 226–238.
[43] A. Papaioannou and K. Magoutis, "Replica-group leadership change as a performance enhancing mechanism in NoSQL data stores," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1448–1453.
[44] C. Melo, J. Dantas, A. Oliveira, D. Oliveira, I. Fé, J. Araujo, R. Matos, and P. Maciel, "Availability models for hyper-converged cloud computing infrastructures," in *Proc. Annu. IEEE Int. Syst. Conf.*, 2018, pp. 1–7.
[45] X. He, L. Ou, C. Engelmann, X. Chen, and S. L. Scott, "Symmetric active/active metadata service for high availability parallel file systems," *J. Parallel Distrib. Comput.*, vol. 69, no. 12, pp. 961–973, 2009.
[46] C. Leangsuksun, et al., "Asymmetric active-active high availability for high-end computing," in *Proc. 2nd Int. Workshop Oper. Syst.*, 2005.
[47] "Apache software foundation," 2017. [Online]. Available: http://zookeeper.apache.org/
[48] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. 41st Int. Conf. Depend. Syst. Netw.*, 2011, pp. 245–256.
[49] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Techn. Conf.*, 2014, pp. 305–319.
[50] "Azure cosmos db," 2017. [Online]. Available: https://azure.microsoft.com/en-us/try/cosmosdb/
[51] G. DeCandia, et al., "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, 2007.
[52] Y. Wang, J. Zhou, C. Ma, W. Wang, D. Meng, and J. Kei, "Clover: A distributed file system of expandable metadata service derived from HDFS," in *Proc. Int. Conf. Cluster Comput.*, 2012, pp. 126–134.
[53] J. Zhou, Y. Chen, X. Gu, W. Wang, and D. Meng, "A virtual shared metadata storage for HDFS," in *Proc. IEEE Int. Conf. Netw. Archit. Storage*, 2015, pp. 265–274.
[54] B. Liskov and J. Cowling, "Viewstamped replication revisited," Comput. Sci. Artif. Intell. Laboratory, MIT, Cambridge, MIT-CSAIL-TR-2012–021, 2012.
[55] K. Arya, R. Garg, A. Polyakov, and G. Cooperman, "Design and implementation for checkpointing of distributed resources using process-level virtualization," in *Proc. Int. Conf. Cluster Comput.*, 2016, pp. 402–412.
[56] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.
[57] "Common open policy service," 2017, [Online]. Available: https://en.wikipedia.org/wiki/CommonOpen_Policy_Service
[58] J. R. Norris, *Markov Chains*," Cambridge, U.K.: Cambridge Univ. Press, 1997.
[59] G. Su, T. Chen, Y. Feng, and D. Rosenblum, "ProEva: Runtime proactive performance evaluation based on continuous-time markov chains," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 484–495.
[60] K. S. Trivedi, "Probability and statistics with reliability, queueing, and computer science applications," Englewood Cliffs, NJ, USA: Prentice-Hall, 1982.
[61] S. He, Y. Wang, Z. Li, X. Sun, and C. Xu, "Cost-aware region-level data placement in multi-tiered parallel I/O systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1853–1865, Jul. 2017.

Jiang Zhou received the PhD degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, in 2014. He is an assistant professor with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include file and storage systems, parallel and distributed computing, metadata management, I/O optimization, and cloud computing.

Yong Chen is an associate professor and director of the Data-Intensive Scalable Computing Laboratory in the Computer Science Department of Texas Tech University. He is also the site director of the NSF Cloud and Autonomic Computing Center at Texas Tech University. His research interests include data-intensive computing, parallel and distributed computing, high-performance computing, and cloud computing. More information about him can be found at http://www.myweb.ttu.edu/yonchen/.



Weiping Wang received the PhD degree in computer science from the Harbin Institute of Technology, China, in 2008. He is a professor with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include database and storage systems.

Shuibing He received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, in 2009. He is now a ZJU100 young professor with the College of Computer Science and Technology, Zhejiang University. His research areas include parallel I/O systems, file and storage systems, high-performance and distributed computing. He has more than 60 papers in major journals and international conferences including the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, ICDCS, IPDPS, ICPP, and CLUSTER.

Dan Meng is a professor and director of the Institute of Information Engineering, the Chinese Academy of Sciences. His main research interests include computer architecture, information security and big data storage and processing.