PAPER

A Distributed and Cooperative NameNode Cluster for a Highly-Available Hadoop Distributed File System

Yonghwan KIM[†a)], ***Nonmember***, **Tadashi ARARAGI**[††], ***Member***, **Junya NAKAMURA**[†††b)], ***Nonmember***, ***and*** **Toshimitsu MASUZAWA**[†c)], ***Member***

SUMMARY Recently, Hadoop has attracted much attention from engineers and researchers as an emerging and effective framework for *Big Data*. *HDFS (Hadoop Distributed File System)* can manage a huge amount of data with high performance and reliability using only commodity hardware. However, HDFS requires a single master node, called a *NameNode*, to manage the entire namespace (or all the i-nodes) of a file system. This causes the *SPOF* (Single Point Of Failure) problem because the file system becomes inaccessible when the *NameNode* fails. This also causes a *bottleneck of efficiency* since all the access requests to the file system have to contact the *NameNode*. Hadoop 2.0 resolves the SPOF problem by introducing manual failover based on two *NameNodes*, *Active* and *Standby*. However, it still has the efficiency bottleneck problem since all the access requests have to contact the *Active* in ordinary executions. It may also lose the advantage of using commodity hardware since the two *NameNodes* have to share a highly reliable sophisticated storage. In this paper, we propose a new HDFS architecture to resolve all the problems mentioned above.

key words: *Hadoop, HDFS, high-availability, distributed NameNodes, automatic failover, load balancing*

1. Introduction

Recently, the size of data that needs to be stored, processed, and maintained is increasing rapidly because of cloud services, social network services, and very many log files. IDC introduces the new term that "digital universe" [1], which is made up of all digital data in the world. And they predict the size of digital universe will be 2.8 ZB ($2.8 * 10^{21}$ Bytes). We usually call these huge data, which are difficult (or impractical) to process with traditional data processing tools or applications, *Bigdata* [4]. Storing and processing *Bigdata* effectively became an emerging issue over the last several years, and Hadoop is one of the most popular frameworks that can handle *Bigdata* effectively [2], [10].

Hadoop is an open source framework for storing and processing large data sets distributedly and effectively on commodity (not highly-reliable) hardware. Surely these hardware are not designed specifically as parts of a large distributed system or storage, but have been appropriated for this role in Hadoop. We can also increase the Hadoop system performance easily by installing additional hardware. For these reasons, many companies or research institutes begin to use Hadoop. For example, NewYork Times uses Hadoop to store all previous articles, and Facebook analyzes 135 TB of datasets everyday. In Japan, Yahoo! Japan uses Hadoop for analysis of access logs, and Rakuten uses it for merchandise managements and behavioral analysis of users. DeNA also uses Hadoop to analyze all user's behaviors whose number is over 2 billions per day.

Hadoop consists of two main components. One is HDFS (Hadoop Distributed File System) [7] and the other one is MapReduce [6]. HDFS is a distributed file system based on GFS (Google File System) [5], and MapReduce is a programming model for processing large data sets.

HDFS consists of only one master node named a *NameNode*, and many worker nodes named *DataNodes*. The NameNode manages metadata of all files and directories of the file system, and DataNodes store actual data blocks in their local storages. Each datum in the file system is divided into several blocks of a predetermined size, and each block is replicated. Because of this replication, HDFS can guarantee reliability of stored data even if some DataNodes fail.

The NameNode maintains *iNodes* which are metadata of all files and directories in HDFS. Each iNode contains a file name, path information, access permission, the number of replications, a list of DataNodes storing actual blocks, and so on. In order to access to HDFS, each client should send requests (commands of the file system) to the NameNode and these requests are processed by the NameNode.

However, HDFS has some problems because the NameNode is the only one master node [3], [18], [19].

1. **Single Point Of Failure (SPOF)**: A failure of the NameNode causes a failure of the entire system.
2. **Namespace Limitation**: The NameNode maintains all iNodes on its local memory in order to process the requests instantly. Thus, the size of the namespace is limited by the size of local memory (RAM) in the NameNode. According to previous works [18], [20], in order to store 100 million files, a NameNode should have at least 60 GB of RAM.

Manuscript received July 28, 2014.
Manuscript revised November 18, 2014.
Manuscript publicized December 26, 2014.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
[††]The author is with Proassist, Ltd., Osaka-shi, 541–0043 Japan.
[†††]The author is with Information and Media Center, Toyohashi University of Technology, Toyohashi-shi, 441–8122 Japan.
a) E-mail: y-kim@ist.osaka-u.ac.jp
b) E-mail: junya@imc.tut.ac.jp
c) E-mail: masuzawa@ist.osaka-u.ac.jp
DOI: 10.1587/transinf.2014EDP7258

Copyright © 2015 The Institute of Electronics, Information and Communication Engineers

3. **Load Balancing Problem**: All requests from clients are received and processed by the NameNode. This may cause the bottleneck of performance.

In this paper, to resolve these problems, we propose a new architecture of HDFS which allows multiple NameNodes. Our proposed system has the following advantages.

- **Resolving SPOF Problem**: We resolve the SPOF problem using multiple NameNodes. When some NameNodes fail, other NameNodes take their roles immediately, instead of the failed NameNodes.
- **Resolving NameNode Limitation**: We can extend the maximum size of the namespace by installing additional NameNodes.
- **Load Balancing**: All iNodes are distributed among many NameNodes. Thus, each NameNode has to manage only a portion of the entire namespace, and it should process only the requests related to the namespace it maintains.
- **Commodity Hardware**: The proposed system can be constructed by only commodity hardware as the original HDFS and needs no special hardware.
- **Weak-linearizability**: The execution of our system guarantees weak-linearizability; which is a weaker property than linearizability [22], [23]. The definition and proof of weak-linearizability will be discussed in Sect. 5.

The rest of this paper is organized as follows: Section 2 introduces some related works As the background of our work, HDFS and ZooKeeper [8] are introduced in Sect. 3. Section 4 presents our proposed system and protocol, and their correctness is shown in Sect. 5. The experimental evaluations of system overhead and performance of load balancing are presented in Sect. 6. A summary and future works are given in Sect. 7.

2. Related Works

From the viewpoint of availaility, the SPOF problem of the NameNode is the most critical problem in HDFS. Therefore, there are various studies of this problem.

In Hadoop 1.x, HDFS introduces a *Secondary NameNode* and a *Backup NameNode* to store the process logs of the NameNode. When the NameNode fails, HDFS can be restored using those stored logs. However, monitoring of faults and restoring of HDFS can be executed only manually by a system administrator.

Hadoop 2.0 introduces a HDFS HA (High-Availability) [11] using two NameNodes which are completely synchronized to resolve the SPOF problem. One NameNode is called an *Active* NameNode, which is responsible for processing the requests from clients. Another NameNode is called a *Standby* NameNode, which is keeping its state synchronized with the Active NameNode to provide a failover. When the Active NameNode fails, the Standby NameNode take over the role of the Active NameNode. However, in order to synchronize these two NameNodes, a shared storage is required between them. Basically the Active NameNode writes its edit logs to this shared storage and the Standby NameNode applies these stored edit logs to its own state. This causes a new SPOF problem because the shared storage is a single point which stores all edit logs. Therefore, reliability of shared storage must be achieved by, for instance a RAID technology or a multiplexing of the network, to guarantee high-availability of HDFS.

In Hadoop 2.1 or later versions, a new architecture, *Quorum Journal Manager* (QJM) [12], is introduced. Hadoop 2.1 also has two NameNodes that are configured and synchronized at all times. But Hadoop 2.1 implements QJM, instead of the shared storage. QJM consists of many machines named *JournalNodes*. The Active NameNode sends its edit logs to the JournalNodes, instead of writing to the shared storage. The Active NameNode's edit logs are durably recorded to a majority of these JournalNodes. The Standby NameNode constantly checks these JournalNodes, and applies the new edit logs (if exists) to its own namespace. The QJM can tolerate up to $\lfloor (N-1)/2 \rfloor$. Note that at least 3 JournalNodes are required and an odd number (i.e. 3, 5, 7, etc.) of JournalNodes are recommended. The QJM resolves the SPOF problem of the shared storage in Hadoop 2.0, but the namespace limitation and the load balancing problems are still left because there is the only one (Active) NameNode in this system.

Facebook also presents an AvatarNode [13], [14] for availability of HDFS, which is keeping its state synchronized with the NameNode. But actually this system does not deal with the namespace limitation or the load balancing problems.

Giraffa file system [15] is proposed to resolve both the namespace limitation problem and the load balancing problem. Giraffa file system adopts *HBase* [16], which is the Hadoop distributed and scalable database and is the open source implementation of Google's BigTable [21]. HBase consists of lots of servers named *RegionServers*, and guarantees scalability linear to the number of RegionServers. In Giraffa file system, all iNodes are stored in HBase in a distributed manner. Therefore, Giraffa can resolve the namespace limitation problem due to the scalability property of HBase. HBase can also process a huge number of clients because HBase processes requests from clients through cooperation among many RegionServers. This implies that Giraffa file system also resolves the load balancing problem.

HBase has only one master node named HMaster. Different from the NameNode in HDFS, the HMaster (in Giraffa file system) does not store any iNode actually, therefore failure of the HMaster does not cause loss of metadata. However, The HMaster is constantly monitoring all RegionServers using heartbeat messages. When the HMaster detects some troubles of RegionServers, it reallocates the data maintained by the failed RegionServers to other RegionServers. Note that the data maintained by the failed RegionServers are stored on local disks periodically. The HMaster

has an important role for availability of the HBase system, and thus the system cannot tolerate failure of the HMaster (even though it does not cause the loss of data). And a failover process of HBase requires a certain amount of time because data of the failed RegionServers are stored in their local disks as the file format called HFile.

3. Background

In this section, we explain the background of our work, HDFS and ZooKeeper, in detail. And we introduce the fault model that we consider in this paper.

3.1 HDFS (Hadoop Distributed File System)

HDFS [7] is an open-source software framework and a logical distributed file system for large data sets. It can store and process large data sets efficiently on clusters of commodity hardware. Clients can access HDFS as an ordinary file system by just sending requests to the NameNode.

HDFS adopts a master/slave model. HDFS consists of one master node named NameNode, and many worker nodes named DataNodes (Fig. 1). The NameNode maintains the entire namespace of the file system, which includes all metadata. Each file is divided into several blocks and each block is replicated. All file blocks (including the replicated blocks) are stored on local disks of DataNodes. Therefore, the files in HDFS are not be lost by DataNodes' failures. Each DataNode reports its own state, which includes a heartbeat, status of blocks, and remaining space of its local disk, to the NameNode periodically.

The NameNode manages all iNodes (metadata) of the HDFS's namespace. Each iNode corresponds to each file or each directory in HDFS and contains all informaiton about its corresponding file, for example, a name, a size, access permissions, and locations of all blocks. The NameNode can process requests from clients with referring these iNodes. The NameNode stores all iNodes on its own memory (RAM) for immediate responces.

If the NameNode fails, clients cannot access to HDFS, because all iNodes are lost. Therefore, fault-tolerance of the NameNode is one of the most critical problem of HDFS's availability, and there are lots of works of this problem.

3.2 ZooKeeper

ZooKeeper [8] is a small cluster that provides highly reliable distributed coordination. All servers in Zookeeper are synchronized, and one of them behaves as a leader (Fig. 2). Zookeeper is distributed over a sets of hosts called an ensemble. As long as majority of the servers are available, the ZooKeeper service is available.

ZooKeeper is organized similarly to a standard file system. It also provides a namespace of the file system. Actually ZooKeeper has only a few commands of the file system and relatively small space, but it can provide high fault-tolerance.

ZooKeeper stores coordination data, e.g. status information, configuration, location information, etc., at *znodes* which are basic storage units. The data stored at each znode, which includes data changes, ACL changes, and timestamp, in a namespace are read and written atomically.

ZooKeeper also provides useful data models as follows.

- **Ephemeral nodes**: Each znode can be a permanent node or an ephemeral node. When a znode is created, the type of node is determined, and is never changed.

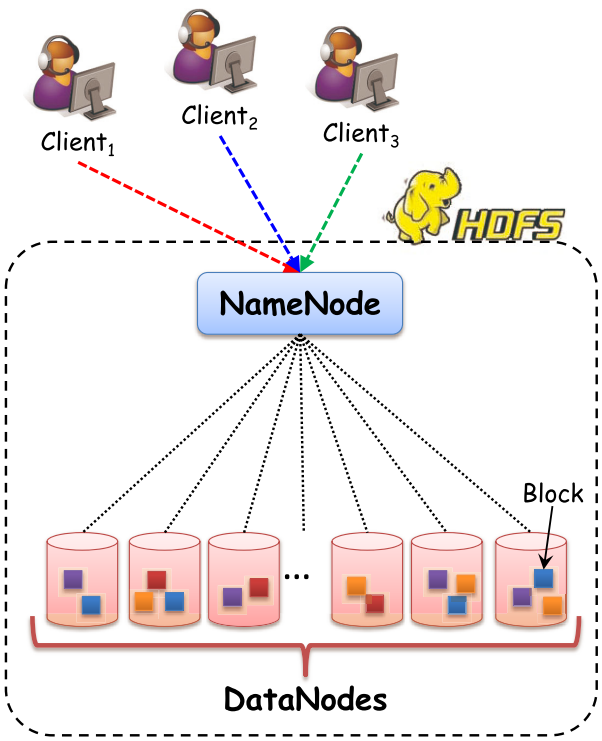


Fig. 1 Architecture of HDFS.

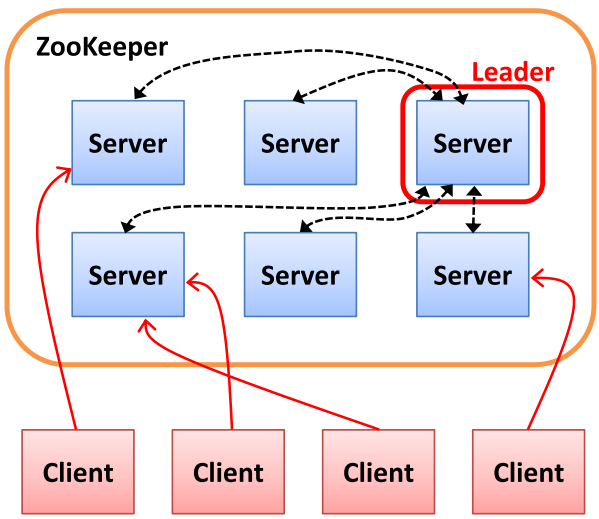


Fig. 2 ZooKeeper service.

In the case of an ephemeral node, the znode exists as long as the session creating the znode is active. This implies that when the client creating the znode fails, the corresponding znode is also deleted. A permanent node is never deleted without a client's explicit request.

- **Watches**: Watches allow clients to get notifications when a znode changes. Watches are set by operations on the ZooKeeper service and are triggered by some events. For instance, a client can register the watcher on a specific znode to detect deletion of the znode. However, Watchers are triggered only once. To receive multiple notifications, a client needs to register the watcher once again.

In our proposed system, all NameNodes are monitored using two data models introduced above. Each NameNode creates a znode on ZooKeeper as an ephemeral node. Some troubles on the NameNodes can be detected because an ephemeral node is deleted when the corresponding NameNode fails. Thus the watcher is registered for each ephemeral node to operate a failover process.

3.3 Fault Model

In this paper, we consider only crash faults of NameNodes, where the faulty NameNodes prematurely stop their operations. We assume that Zookeeper is reliable and can eventually detect the faults of NameNodes.

4. Our Approach: Cooperative Distirbuted NameNode Cluster

4.1 Namespace Partitioning

As we mentioned in the previous section, all iNodes are stored on the memory of the NameNode. An iNode includes all information (name, size, access permission, block locations, etc.) of its corresponding file (or directory). The namespace of HDFS can be represented by the set of iNodes.

Our proposed system has several NameNodes. It partitions the namespace of HDFS and stores replicas of the fragments distributedly among the NameNodes. Figure 3 describes the difference between our system and Hadoop 2.0 [11]. A tree on the upper-left shows the entire namespace. The root of the tree corresponds the root directory, and the sub-directories under the root directory are described as subtrees (triangles). To help to understand, identifiers of each subtree is attached as $NS_1, NS_2, NS_3, \ldots$.

Hadoop 2.x has two NameNodes, Active and Standby. Each of the NameNodes stores the entire namespace. The Active NameNode is responsible for processing requests from all clients, and it writes edit logs to a shared storage. When new edit logs are written on a shared storage (or JournalNodes in Hadoop 2.2 or later versions), the Standby NameNode reads those edit logs and applies them to its own state. Therefore, two NameNodes are completely synchronized at all times, and this enables the Standby NameNode

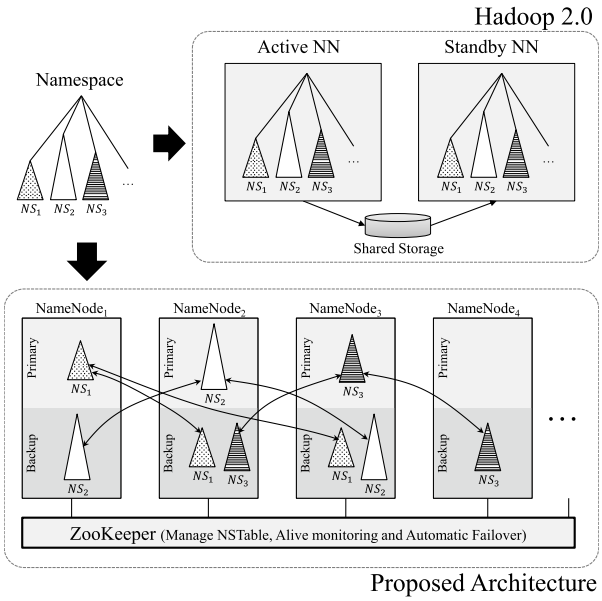


Fig. 3 Namespace partitioning.

to operate as an Active NameNode when the (original) Active NameNode fails.

In contrast, the lower part of Fig. 3 presents our proposed architecture. Our system contains N NameNodes denoted by $NameNode_1, NameNode_2, \ldots, NameNode_N$, and this set of NameNodes are called a **NameNode cluster**. The namespace is partitioned into m disjoint fragments $(NS_1, NS_2, \ldots, NS_m)$. The fragments are replicated and distributedly stored in the NameNodes. In Fig. 3, each fragment has 3 replicas including itself. These 3 replicas are stored on different NameNodes, for example, replicas of NS_i are stored on $NameNode_1$, $NameNode_2$, and $NameNode_3$.

For each fragment N_i, the NameNodes storing a replica of N_i are classified into a **Primary NameNode** and **Backup NameNodes**. Only one NameNode can be a Primary NameNode of NS_i and the Primary NameNodes of different fragments N_i and N_j are determined independently. Only the Primary NameNode of N_i can process the requests on NS_i from clients. All the other NameNodes are **Backup NameNodes** modify NS_i they store if the Primary NameNode allows.

We deal with the SPOF problem of HDFS using these NameNodes. When the Primary NameNode of NS_i modifies the state of NS_i, it sends its edit logs to all Backup NameNodes storing NS_i. When the Primary NameNode of NS_i fails, one of the Backup NameNodes takes over the role of the Primary NameNode. This failover process can be operated within a short time. In the lower part of Fig. 3, the Primary NameNode of NS_2 is $NameNode_2$, and NS_2 is also maintained by $NameNode_1$ and $NameNode_3$. When $NameNode_2$ fails, $NameNode_1$ or $NameNode_3$ becomes the Primary NameNode of NS_2.

Certainly, it is not easy to guarantee that the states of all replicated NS_i are completely synchronized to be identical,

because message communication in the distributed systems might be delayed unpredictably. Therefore we propose the majority-based protocol in Sect. 4.2.

All fragment information, which includes the partition information and the address of the Primary NameNode of each fragment, is maintained by ZooKeeper, a highly-reliable distributed coordinator. We call this information **NSTable** (NameSpace Table), and we assume that clients can recognize the destination address of each request referring the NSTable.

4.2 Cooperative Process

In this Section, we explain how NameNodes cooperate.

4.2.1 Consistency Mechanisms

As mentioned in Sect. 4.1, in our proposed architecture, we partition a namespace into m fragments, replicate each fragment to make k replicas, store these replicas in k different NameNodes (one Primary NameNode and k-1 Backup NameNodes). We call k *redundancy factor*. The redundancy factor determines a degree of fault-tolerance.

In our architecture, synchronization among NameNodes is required to update each fragment consistently. The synchronization is realized using message communication implemented by RPC (Remote Procedure Call) protocol. Notice that a message sender never knows whether a message is received or not, until its acknowledgement comes back.

Now we introduce a majority-based message protocol that can make the replicas of each fragment keep consistency. This majority-based protocol can guarantee the consistency even if up to ($\lfloor \frac{k-1}{2} \rfloor$) NameNodes fail, where k is the redundancy factor.

Our Majority-based Protocol: To send requests to HDFS, a client should send the requests to the Primary NameNode. The clients can get the address of the Primary NameNode from ZooKeeper. When the Primary NameNode receives a request from a client, it checks whether the request is valid or not. The request is invalid when the client gets an out-of-date address of the Primary NameNode. This happens because NSTable can be changed by the failover process. If the request is valid, the Primary NameNode processes (applies) the request.

A request from a client may modify the state of the fragment. If the state of the fragment is modified, the Primary NameNode sends *sync* messages to all Backup NameNodes maintaining the replicas of the fragment. The Backup NameNode which receives the *sync* message sends *ack* message to the Primary NameNode. The Primary NameNode has to wait for receipts of the *ack* messages from a majority of the Backup NameNodes.

There are k replicas of each fragment and $(k-1)$ Backup NameNodes maintain the replicas, thus, the Primary NameNode requires $\frac{k}{2}$ or more *ack* messages to proceed to the next step. If a majority of the *ack* messages are received by the Primary NameNode, it can fix the modificiation of the fragment's state (i.e., commit the transaction) and sends the result (response) of the request to the client. Finally, the Primary NameNode sends *update* messages to all Backup NameNodes to fix the replicas' states of the fragment.

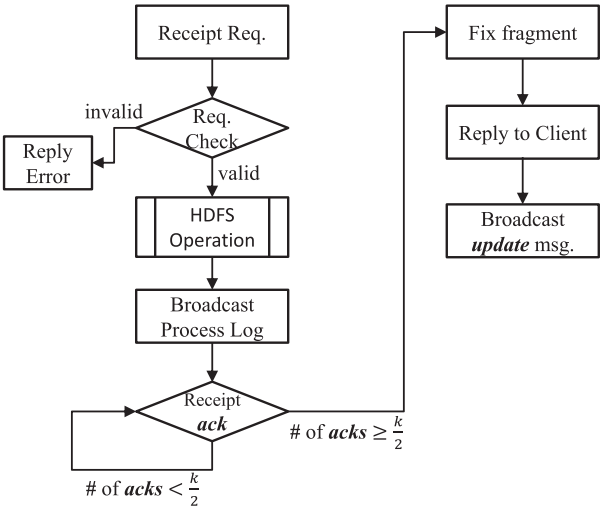


Fig. 4 Process of the request on a primary NameNode.

Figure 4 shows the flow of processing a request from a client on the Primary NameNode. We describe this process in detail through Algorithms from 1 to 3. In this Algorithms, we represent the system's state like NS_f^v, where v means the version of current view which is incremented by the failover process and f means the version of its own fragment (f can be increased by processing the request). We also represent the Primary NameNode just as *PNN*.

4.2.2 Automatic Failover

In our architecture, we use ZooKeeper, which is a distributed coordinator, in order to keep consistency among NameNodes. Each NameNode creates a session that cre-

Algorithm 1 When a request from a client is received

```
1:  procedure OnRequest(Req_j)          ▷ when Req_j from Clt_i is received
2:      if !isValid(NSTable, Req_j) then                    ▷ validity check
3:          Notify Clt_i of the invalid request   ▷ Clt_i may check NSTable
4:          return
5:      end if
6:      if update of NS_f^v is not necessary then
                ▷ Some requests do not require the update of NS (e.g. ls)
7:          return Response of Req_j
8:      end if
9:      Create NS_(f+1)^v from the latest state NS_f^v (by applying Req_j)
10:     Set NS_(f+1)^v as a tentative state
11:     Send Sync_(f+1)^v (edit log for NS_(f+1)^v) to backup NameNodes
12:     Wait until the receipt a majority of the ack messages
13:     Update NS_(f+1)^v to a fixed state
14:     Send Result (Req_j) to Client(Clt_i)
15:     Send Update_(f+1)^v (for log_(f+1)^v) to backup NameNodes
16: end procedure
```

Algorithm 2 When *Sync* is received

```
1: procedure OnLog(Sync^{v'}_{f'})          ▷ when Sync from PNN is received
2:     if isPNN then                                  ▷ PNN received Sync
3:         if v' > v then          ▷ comparing log's view with current view
4:             Check ZooKeeper's Failover Log
5:             Sync its own state to the latest log version
6:         end if
7:     end if
8:     if (v = v') & (f+1 = f') then      ▷ next version of fragment's state
9:         Create NS^{v}_{(f+1)} as unfixed state (by applying Sync^{v'}_{f'})
10:        Send ack^{v}_{(f+1)} to PNN
11:    end if
12: end procedure
```

Algorithm 3 When *Update* from PNN is received

```
1: procedure OnUpdate(Update^{v'}_{f'})
2:     if NS^{v'}_{f'} is unfixed state then                ▷ valid update?
3:         Update NS^{v'}_{f'} to fixed state
4:     end if
5: end procedure
```

ates znode as an ephemeral node on the ZooKeeper and sets a watcher on that znode. This process is used for detecting a fault of the NameNode. When a NameNode fails, the ephemeral node created by that NameNode is deleted due to disconnection of the session with it. If the Primary NameNode fails, the ZooKeeper notifies all Backup NameNodes of the Primary NameNode's failure. The next Primary NameNode is determined by rotation referring NSTable.

The Backup NameNode which received the notification from the ZooKeeper records its state of the managed fragment to the ZooKeeper. The next Primary NameNode checks the ZooKeeper and waits until a majority of the replicas' states ($\frac{k}{2}$ or more) are recorded. When a majority of the fragment's states is recorded on the ZooKeeper, the Primary NameNode elects the latest state of the fragment as the state after the failover. NSTable should be modified during this failover process.

We should also consider the following situations:

1. An ephemeral znode created on the ZooKeeper may be deleted even when the NameNode does not fail. This is caused by some temporary network troubles.
2. During a failover, the next Primary NameNode may also fail before the termination of the failover.

In case 1, the (previous) Primary NameNode which is excluded by some temporary network troubles can rejoin to the NameNode cluster with creating a new ephemeral node and setting a watcher once again. To guarantee consistency of the namespace, process log of the failover is recorded on the ZooKeeper. This log can be used when the previous NameNode recognizes the disconnection and returns to the system.

In case 2, the ephemeral node created by the next Primary NameNode is deleted although a failover is underway. A new failover is initiated by the ZooKeeper and the new next Primary NameNode coordiates it.

A Failover Process: As we explain above, each NameNode creates an ephemeral znode on the ZooKeeper and sets a watcher on it for a failover.

When the ZooKeeper finds deletion of the ehpemeral node, it checks whether it is created by the Primary NameNode or not. If it is created by the Primary NameNode, the ZooKeeper sends *Suspect* messages to all Backup NameNodes. The Backup NameNode which received the *Suspect* message, it records its fragment's state on a predetermined znode on the ZooKeeper. The next Primary NameNode waits until a majority of the fragment's states is recorded on the ZooKeeper, and adopts the latest state as its own state. The next Primary NameNode sends *consensus* messages to all Backup NameNodes, and waits until a majority of the Backup NameNodes. Finally, the next Primary NameNode becomes the Primary NameNode, and sends *Resync* messages to all Backup NameNodes. This Primary NameNode can begin to process the request from clients immediately, because all Backup NameNodes must receive the *Resync* message before receiving any *Sync* messages from the new Primary NameNode due to FIFO property.

We describe this process in detail through Algorithms from 4 to 7.

Algorithm 4 When *Suspect* from ZooKeeper is received

```
1: procedure OnSuspect(NN_x)
2:     Record its latest NS^{v_i}_{f_i} to znode on the ZooKeeper
3:     if self is the next PNN then
4:         Wait until a majority of the NS^{v}_{f} are recorded
5:         nextNS ← newest one among collected NS^{v}_{f}s
6:         Send Consensus(nextNS) to backup NameNodes
7:         Wait until a majority of the Ready are received
8:         Broadcast Resync(nextNS) to NameNodes (include itself)
9:         Write Failover's Log to the ZooKeeper
10:    end if
11: end procedure
```

Algorithm 5 When *Consensus* is received

```
1: procedure OnConsensus(nextNS)
                          ▷ when Consensus from the next PNN is received
2:     if isNextPNN then                          ▷ sender is next PNN?
3:         Synchronize its own state to nextNS
4:         Send Ready() to the next PNN
5:     end if
6: end procedure
```

Algorithm 6 ZooKeeper's Watcher Code

```
1: procedure OnDisconnect(NN_x)
          ▷ when the session with the NameNode(NN_x) is disconnected
2:     if NN_x is the Primary NameNode then
3:         Send Suspect to NameNodes maintaining PNN's replica
4:     end if
5: end procedure
```

Algorithm 7 When a NN finds disconnection with ZK

```
1: procedure OnDisconnect() ▷ when the session w/ ZK is disconnected
2:     Try to start session w/ ZK and makes a new ZNode
3:     Check the failover's Log and synchronize with the current PNN
4:     Set Watcher to ZK
5: end procedure
```

4.3 Implementation Sketch

In this Section, we present details of implementation of our system to some extent.

Namespace Table: The namespace table (*NSTable*) is a hash table which has a file path as an input and it returns the corresponding fragment number and Primary NameNode's address as outputs. NSTable is stored in the znode of ZooKeeper, a highly reliable distributed file system. We assume that NSTable is never lost due to the high fault-tolerance of ZooKeeper. NSTable includes the NSTable's version which is updated at each failover process. All NameNodes and clients can get the most recent NSTable from ZooKeeper at any time. Note that NSTables' versions some NameNodes or clients get can be different.

ZooKeeper: ZooKeeper is a highly reliable file system. In our system, ZooKeeper has two main roles.

First, we uses ZooKeeper to store NSTable reliably. The most recent NSTable is always stored in the predetermined znode of ZooKeeper. Our system communicates with ZooKeeper to update NSTable only. All NameNodes and clients can get the most recent NSTable from ZooKeeper.

Another role of ZooKeeper is to detect faults of NameNodes. ZooKeeper is a distributed file system consisting of znodes, but it can be utilized to detect nodes' crash faults using ephemeral nodes. Each NameNode in our system creates its own ephemeral node in ZooKeeper. When a NameNode fails, its corresponding ephemeral node is deleted by a property of ZooKeeper, and some predetermined actions (a failover process in our system) is operated (Algorithm 6).

ZooKeeper is operated separately from our system and requires no modification for application to our system. Our system uses ZooKeeper as a small and reliable file system for storing NSTable. And NameNodes' crash faults can be detected by ZooKeeper using their ephemeral nodes in ZooKeeper system. Therefore, new (or recovered) NameNodes have to create their ephemeral nodes and set watchers (Algorithm 6) for failover processes.

NameNodes: A client sends a request to the file system using RPC (same as the original HDFS). Each NameNode creates a new (java) thread and executes Algorithm 1 when it receives a request from the client.

Each NameNode maintains the set of the NameNodes' addresses of each fragment. It knows from the NSTable which NameNode is the Primary NameNode in this set. Therefore, each NameNode can check the validity of the request from the client: it detects invalidity of the request when it is not the Primary NameNode.

Communications among NameNodes are implemented by Remote Procedure Call (RPC) which is provided by Hadoop library. A Primary NameNode can send the corresponding fragment's update information to all Backup NameNodes using RPC.

Clients: Each client can send requests (commands to the file system) to our system, which is the same as the original HDFS. However, as explained in this section, each client has to know the corresponding Primary NameNode's address which it sends a request to. This implies that each client requires NSTable to decide the target NameNode.

In our system, each client has NSTable instead of NameNode's address in the original HDFS. Each client can easily get the most recent NSTable from ZooKeeper. If a client sends a request to an invalid Primary NameNode because of an outdated version of NSTable, it is notified from the NameNode which received the request.

5. Proof of Correctness

In this Section, we prove the correctness of our majority-based protocol and failover process.

Our system can guarantee the follows regardless of some NameNodes' failure.

1. If a client receives a reply of its request from the system, that the request is commmitted.
2. When a client refers a namespace in HDFS, it never sees an older namespace than the one it refers before.
3. A namespace in HDFS is modified by clients only. This implies the system never changes a namespace internally.

5.1 Preliminaries

In this Section, to help to understand our proof, we explain some conceptional notations of the events that can be occurred in our proposed system.

To access HDFS, each client sends a request to the Primary NameNode and waits until a result is back. Following a shared memory model of distributed systems [22], we call these events an invocation and a result respectively. A invocation (Inv) represents a sending of a request, and a result (Res) represents a receipt of a response from a system.

Each Inv and Res has an index, like Inv_i or Res_j. Res_i is the result of Inv_i. A pair $Tx_i = (Inv_i, Res_i)$ of an invocation and the corresponding result is called a **transaction**.

Figure 5 describes interaction between a client and the system. A client sends a request (Inv_i) to an interface process of the system, and receives a response (Res_i) from the interface process. When the interface process receives Inv_i, it forwards it to the system (the Primary NameNode) and forwards Res_i to a client when received from the system. We call Get_i an event that the Primary NameNode receives Inv_i, and call $Send_i$ an event that the Primary NameNode

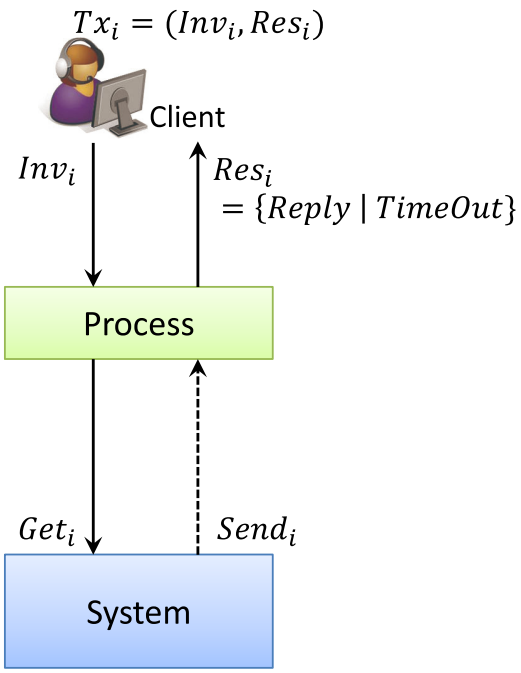


Fig. 5 Representations of a request.

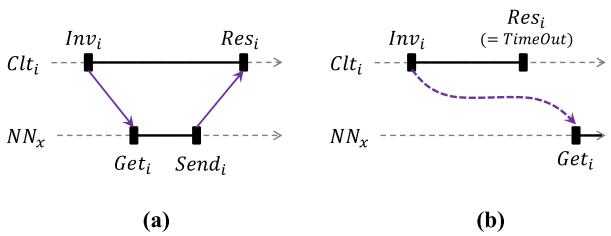


Fig. 6 Timings of *Inv* and *Res*.

sends a result to the interface process.

We can summarize these events as follows: a client sends a request to the system (Inv_i), the Primary NameNode receives Inv_i (Get_i), the Primary NameNode sends a response back to the client ($Send_i$), finally, the client receives the response (Res_i). The causal order of these events becomes as follows inevitably: $Inv_i \prec Get_i \prec Send_i \prec Res_i$.

However, in our proposed system, the system might not send the response when failures or delays occur in the system. If the response is not received by the client within a certain interval time, the client considers its own request is failed. We call this $TimeOut$. As a result, all transactions of clients can be presented to $Tx_i = (Inv_i, Res_i)$, where Res_i can have two kinds of result, *Reply* or $TimeOut$.

Figure 6 shows events of a request using a time-space diagram. If there is no fault in the system, a client Clt_i and a NameNode NN_x may operate like Fig. 6 (a). We can find the causal order of the events $Inv_i \prec Get_i \prec Send_i \prec Res_i$.

However, the client cannot receive the response within a predetermined interval time, Res_i becomes not *Reply* but $TimeOut$. Figure 6 (b) illustrates the case of $Res_i = TimeOut$ because of a message's delay. Note that, in this case, only the causal orders $Inv_i \prec Res_i$ and $Inv_i \prec Get_i \prec Send_i$ (if exist) hold.

In this paper, we model our system as a state machine. A request from a client is regarded as an input of this state machine, the state of the system is changed deterministically by this input. This implies that our system's state is described by a sequence of requests. If a Tx_i is in an input sequence, we call that request Tx_i is *applied*. In our system, the relation between the system state and the sequence of requests is complicated because of concurrency by multiple NameNodes, NameNode fauls and message delays. We explain the details of the relation between an input sequence and a system's state.

5.2 Definitions

In this Section, we introduce some definitions for our proof.

The guiding principle of our protocol is that there is only one Primary NameNode for each fragment at any time, however some NameNodes may temporarily recognize a different NameNode as a current Primary NameNode because of asynchrony. To avoid ambiguity, we define follows:

Definition 1. Global Primary NameNode. *A Global Primary NameNode at a certain moment is the NameNode which sends a Resync message lastly. If there is no NameNode which has sent a Resync message, the initial Primary NameNode becomes a Global Primary NameNode.*

We define a state of a NameNode as follows.

Definition 2. A State of a NameNode. *A state of a NameNode can be determined by a sequence of requests. We represent the state of the NameNode NN_i as follows: $\sigma_{NN_i} = [Req_i \prec Req_j \prec \ldots]$.*

A state of a NameNode starts with its initial state, and should be changed by requests from clients deterministically. Therefore, a current state of the Primary NameNode can be specified by a sequence of corresponding *Get* events and the Backup NameNodes can be specified by a sequence of corresponding $Sync$ events from the Primary NameNode.

Each Req_i in a sequence σ_{NN_i} has one of between the two states, fixed or unfixed. We explain a fixed or unfixed request of each Req_i as follows.

- Each NameNode adds an **unfixed** Req_i to the end of its own state, when it (if it is the Primary NameNode) receives the corresponding Get_i, or it (if it is the Backup NameNode) receives the corresponding $Sync(Get_i)$ message from the Primary NameNode.
- Each unfixed Req_i is changed to a **fixed** request, when the NameNode (if it is the Primary NameNode) receives corresponding *ack* messages from a majority of Backup NameNodes, or the NameNode (if it is the Backup NameNode) receives the corresponding $Update(Get_i)$ message from the Primary NameNode.
- After a failover, each NameNode may receive the

Consensus(σ_{NN_c}) message, where σ_{NN_c} is the consensus state of failover process, and update its state to the received state σ_{NN_c} to its own state. Some *Req* in only its own state may be deleted by this synchronization, however, a fixed *Req* is never deleted by this synchronization (we will prove it in this Section). All *Req* in the σ_{NN_c} becomes **fixed** when each NameNode receives *Resync*().

The notation $\overline{\sigma}_{NN_i}$ indicates the subsequence of σ_{NN_i} consisting of all fixed *Req*. And we call $\overline{\sigma}_{NN_i}$ the fixed state of the NameNode NN_i.

Definition 3. A state of the system. *A state of the system σ_S is a state of the Global Primary NameNode at the time.*

We represent a fixed state of the system $\overline{\sigma}_S$ consisting of all fixed *Req* in σ_S.

We say that the request is **processed** or **applied** if the corresponding Req_i is included in σ_S.

Definition 4. History. *A history H is a sequence consisting of Inv and Res.*

We can construct a history H which consists of all *Inv* and *Res* events, and we represent this history the **global history**, H_G. All *Inv* and *Res* events in H_G are sorted in chronological order. Figure 7 shows an example of execution of 3 clients. In this case, we can construct the Global History $H_G = [Inv_{i1}, Inv_{k1}, Inv_{j1}, Res_{j1}, Res_{i1}, Inv_{i2}, \ldots]$.

We can also construct a subsequence of H_G which consists of all *Inv* and *Res* events that are performed by the client Clt_i, and we represent this subsequence $H_G^{(Clt_i)}$. In the case of Fig. 7, $H_G^{(Clt_j)}$ becomes $H_G^{(Clt_j)} = [Inv_{j1}, Res_{j1}, Inv_{j2}, Res_{j2}]$.

Definition 5. Sequential History. *A history H is a sequential history, if (1) the first event of H is an Inv (2) Each Inv is immediately followed by a matching Res, and each Res immediately follows a matching Inv.*

A sequential history represents a behavior of a client in the absence of concurrency. Each client sends a request (Inv_i), and receives a response ($Res_i = Reply$) or treat the request as a failure ($Res_i = TimeOut$). A client sends the next request after *Res* is decided, therefore $H_G^{(Clt_i)}$ becomes a sequential history.

If a history H is a sequential history, it can be expressed using corresponding transactions as follows. $H_G^{(Clt_i)} = [Inv_0, Res_0, Inv_1, ..Res_n] = [Tx_0, Tx_1, ..Tx_n]$

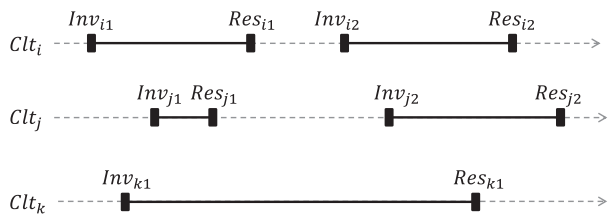


Fig. 7 An example of the execution of 3 clients.

Sequential Specification: A sequential specification consists of a set of operations and a set of sequences of operations [22]. A sequential specification is defined in detail by the system specifications.

On the shared object S, a sequential specification of S determines the total set of histories such that each Res_i in the sequential history $H = [Inv_1, Res_1, Inv_2, Res_2, \ldots]$ has the result when the system processes the request $[Inv_1, Inv2, \ldots, Inv_i]$ sequentially. If the (sequential) history H of S is in a sequential specification of S, we call H a **legal** sequential history.

Definition 6. Linearizability. *A global history H_G is linearizable if there exists a permutation π of all the operations (Inv and Res) in H_G such that*

1. *π is a legal sequential history, and*
2. *if the Res_a occurs in H_G before Inv_b, then Inv_a appears before Inv_b in π.*

Linearizability can guarantee the strong consistency for concurrent objects [22]–[24]. Linearizability also guarantees a local property which implies that the system consisting of linearizable objects is also linearizable.

In the case of our proposed system, Res_i may have *TimeOut* instead of *Reply* due to failures of some NameNodes or unpredictable delay. From the definition of a sequential specification, a history including *TimeOut* never is a legal sequential history. Thus we define a new property, **weak-linearizability** as follows.

Definition 7. Weak-Linearizability. *A global history H_G is weakly-linearizable if there exists a permutation π of a subset of the operations (Inv and Res) in H_G such that*

1. *π can be a legal sequential history by replacing all TimeOuts with approprite Replys.*
2. *All transactions $Tx_i = (Inv_i, Res_i = Reply)$ are in π.*
3. *If $Res_a = Reply$ occurs in H_G before Inv_b, then Inv_a appears before Inv_b in π (if Inv_b is in π).*
4. *If Res_a occurs before Inv_b in $H_G^{(Clt_x)}$, then Inv_a appears before Inv_b in π (if both Inv_a and Inv_b are in π).*

Weak linearizability is a weaker property than linearizability: weakly-linearizable global history H_G is linearizable if H_G contains no transaction $Tx = (Inv, Res)$ such that $Res = TimeOut$.

Conditions 1 and 2 of Definition 7 imply that all the transactions with $Res = Reply$ are committed but some of the transactions with $Res = TimeOut$ can be aborted. Notice that Condition 1 allows some transactions with $Res = TimeOut$ can be committed. In this case, *TimeOut*s occur at the clients but the transactions are committed by the system.

Condition 3 of Definition 7 means that the processing order of the transactions with $Res = Reply$ is preserved even if the requests are from different NameNodes. Condition 4 of Definition 7 implies that the processing order of the transactions with $Res = TimeOut$ is ensured only on the same NameNode.

To help to understand the definitions we show an exmaple. We assume two requests, *mkdir* and *mv*, both requests are provided in ordinary file systems. $mkdir([target])$ is the request to make a directory $[target]$, and $mv([a],[b])$ is the request to move $[a]$ to $[b]$. The results of these requests can be *ok* or $fail$. *ok* means that request is processed successfully, and $fail$ is not for some reasons, for example, there is no $[a]$ in the file system.

We show 4 executions (from H_{G1} to H_{G4}) using the above two requests with two clients, Clt_i and Clt_j. Each Inv and Res contains its client's identifier and request number.

$$\begin{aligned}
H_{G1} = {} & Inv_{i1}(mkdir(/a))\ Res_{i1}(ok)\ Inv_{j1}(mv(/a,/b)) \\
& Inv_{i2}(mv(/a,/c))\ Res_{j1}(ok)\ Res_{i2}(fail(no\ /a)) \\
H_{G2} = {} & Inv_{i1}(mkdir(/a))\ Res_{i1}(ok)\ Inv_{j1}(mv(/a,/b)) \\
& Inv_{i2}(mv(/a,/c))\ Res_{j1}(fail(no\ /a)) \\
& Res_{i2}(fail(no\ /a)) \\
H_{G3} = {} & Inv_{i1}(mkdir(/a))\ Res_{i1}(TimeOut)\ Inv_{j1}(mv(/a,/b)) \\
& Inv_{i2}(mv(/a,/c))\ Res_{j1}(fail(no\ /a)) \\
& Res_{i2}(fail(no\ /a)) \\
H_{G4} = {} & Inv_{i1}(mkdir(/a))\ Res_{i1}(TimeOut)\ Inv_{j1}(mv(/a,/b)) \\
& Inv_{i2}(mv(/a,/c))\ Res_{j1}(fail(no\ /a)) \\
& Res_{i2}(fail(no\ /a))\ Inv_{i3}(mv(/a,/d))\ Res_{i3}(ok)
\end{aligned}$$

In H_{G1} and H_{G2}, after *mkdir* of Inv_{i1} is processed successfully. The requests invoked by Inv_{i2} and Inv_{j1} are processed concurrently In H_{G1}, the request of Inv_{i2} becomes $fail$ and the request of Inv_{j1} is processed correctly. This means that the directory $/a$ has changed to $/b$ by the request of Inv_{j1} before the request of Inv_{i2} is processed. As a result, H_{G1} can be represented as $\pi = Tx_{i1}Tx_{j1}Tx_{i2}$ and linearizable.

In H_{G2}, the requests invoked by Inv_{i2} and Inv_{j1} are sent concurrently and both of them become $fail$. For this result, we can assume that the request of Inv_{i1} is processed after them like $\pi = Tx_{j1}Tx_{i2}Tx_{i1}$. In this case, the requests of Inv_{i2} and Inv_{j2} are failed because there is no $/a$. However, Inv_{i2} (or Inv_{j1}) is followed by Inv_{i1} in π, but $Res_{i1} \prec Inv_{i2}$ in H_{G2}. This violates the condition of linearizability (Condition 2 of Definition 6). Therefore H_{G2} is not linearizable.

H_{G3} and H_{G4} are histories including $TimeOut$. H_{G3} is a minor change of H_{G2}, Res_{i1} is changed from *ok* to $TimeOut$. In this case, we can delete $Tx_{i1} = (Inv_{i1}, Res_{i1})$ becuase Res_{i1} has $TimeOut$. Therefore, we can construct $\pi = Tx_{j1}Tx_{i2}$, and this means H_{G3} is weakly-linearizable.

H_{G4} can be made as adding some operations to the end of H_{G3}. A new invocation Inv_{i3} is a request of *mv*, and it is processed correctly. This means that Inv_{i1} is applied successfully, thus, we can not delete Inv_{i1} from π even if Res_{i1} is $TimeOut$. According to condition 4 of Define 7, the order $Inv_{i1} \prec Inv_{i2}$ has to be ensured regardless of Res_{i1}. However, Res_{i2} is failed due to the absence of $/a$. Therefore, H_{G4} is not weakly-linearizable.

Weak-linearizability can guarantee consistency but it cannot be determined whether a transaction is committed or not when its reply is $TimeOut$. In this case, the transaction with $TimeOut$ can be either committed or aborted. Even if a request is timed out, a client can check whether the request is committed or not by accessing the HDFS. Certainly the timed out request is never applied after checking HDFS, this property is practical enough.

Now we prove that our proposed system can guarantee weak linearizability regardless of some faults in the next Section.

5.3 Correctness of Our System

In this Section, we prove the consistency of our system by showing the following theorem.

Theorem 1. *Any global history created by our system is weakly-linearizable.*

We consider any execution ϵ of our proposed system and its global history. To help to prove, we construct a sequence consisting of all Gets and $Send$s in ϵ. We represent this sequence H_S. The global history H_G consists of the events of the clients, but H_S consists of the events of the NameNodes.

Now we introduce how to construct a sequential history π from H_G by referring H_S as follows.

STEP 1: To construct a history $\overline{H_G}$ from H_G, arrange all Inv events in H_G in the order of their corresponding Get events in H_S. Since a Get_i event occurs after an Inv_i event, each Get event in H_S has its corresponding Inv event in H_G. If an Inv event has no corresponding Get event, delete it and its corresponding Res in H_G. After that, move each Res event in H_G so that it comes immediately after its corresponding Inv.

This process makes $\overline{H_G}$, a sequential history.

STEP 2: Remove from $\overline{H_G}$ all transactions satisfying (a) Res is $TimeOut$ and (b) corresponding fixed Req never appear in $\overline{\sigma_S}$ for all time.

Let π be a sequential history constructed by STEP 1 and 2. Now we show π satisfies all conditions (1 to 4) of Definition 7.

At first, we introduce the following lemma to prove this sequential history is legal.

Lemma 1. *If Get_i procedes Get_j in H_S, the corresponding Req_i also procedes Req_j in σ_S at any time if both of them are in σ_S.*

Proof of Lemma 1. (Proof by contradiction) Assume the system consists of N NameNodes, NN_1 to NN_N, and they manage the same replicas. NN_1 is the Primary NameNode, the NameNode with the next index should become the next Primary NameNode at each failover.

Moreover, assume that NN_N is never failed during the execution of the system. This assumption is valid without loss of generality because our proposed system guarantees $\lfloor \frac{k-1}{2} \rfloor$-fault tolerance and this implies some fault-free NameNodes are left.

Now we can choose any two Get events, $Get_i \prec Get_j$,

in H_S. And assume that two corresponding Req_i and Req_j appear in reverse order ($Req_j \prec Req_i$) in σ_S, which means $Sync(Get_j)$ occurrs before $Sync(Get_i)$ on NN_N.

If Get_i and Get_j occur on the same NameNode, $Sync(Get_i)$ must precede $Sync(Get_j)$ because of FIFO property. Therefore, we only consider the case that these events occur on different NameNodes, say NN_i and NN_j respectively.

Each NameNode applies a received $Sync()$ message only when its sender is a Primary NameNode (possibly different from the Global Primary NameNode). To recognize that a specific NameNode is the current Primary NameNode, each NameNode has to receive a $Resync()$ message from that NameNode. From this protocol, each NameNode can apply a $Sync()$ message only when it receives $Resync()$ message from the Primary NameNode before the $Sync()$ message. Therefore, NN_N has to receive $Resync(NN_i)$ before $Sync(Get_i)$ to apply Get_i to its state. As a result, in order to apply Get_i after Get_j, the order of the events on NN_N should be as follows:

$$Resync(NN_j) \prec Sync(Get_j) \prec Resync(NN_i) \prec Sync(Get_i)$$

$Resync()$ must be sent before sending of $Sync(Get_i)$ on NN_i (NN_j also). And $Sync(Get_i)$ precedes $Sync(Get_j)$ by the assumption. Thus the following three cases of the sending order are considerable without any violations of causality.

1. $Resync(NN_j) \prec Resync(NN_i) \prec Sync(Get_i) \prec Sync(Get_j)$
2. $Resync(NN_i) \prec Resync(NN_j) \prec Sync(Get_i) \prec Sync(Get_j)$
3. $Resync(NN_i) \prec Sync(Get_i) \prec Resync(NN_j) \prec Sync(Get_j)$

In the case 1, NN_j becomes the Primary NameNode before NN_i do due to the total order of $Resync()$ events. NN_N receives $Resync(NN_j)$ before $Sync(Get_j)$, and this makes $\sigma_N = [\sigma_j, Req_j]$ (σ_j is the NameNode state delivered by a $Consensus$ message before the $Resync$ message). After that, $Resync(NN_i)$ is received for the later failover, σ_N is synchronized with σ_i, which is concensus state by NN_i, as a result. But Req_j is not in σ_i, because a consensus of σ_i is already finished before occurring Get_j. Therefore, σ_N never applies Req_j (Get_j event) to its own state. This contradicts the assumption.

In the cases 2 and 3, NN_i becomes a Primary NameNode before NN_j do. This means $Resync(NN_j)$ becomes the later failover than $Resync(NN_i)$. However, NN_N receives $Resync(NN_j)$ before $Resync(NN_i)$ by the assumption. Therefore, $Resync(NN_i)$ cannot be applied because it has an older failover version than $Resync(NN_j)$, and thus Req_i (Get_i event) is never applied. This also contradicts the assumption.

As a result, all Reqs which are applied to σ_S preserve the order of corresponding Gets in H_S. □

From Lemma 1, all Reqs are applied to σ_S as the same order of corresponding Gets in H_S (if they are applied). Thus the arrangement of all events in H_G (STEP 1) is valid because all requests are applied to σ_S at any time in the same order as that of Get events in H_S (if they are applied).

Now we show that deletion of some events in $\overline{H_G}$ (STEP 2) makes a legal sequential history.

Lemma 2. *Once any Req_i in σ_S becomes fixed, it keeps its state and is never deleted.*

Proof of Lemma 2. We can prove Lemma 2 using the property of majority.

In order to Req_i is fixed, it is necessary (a) to receive corresponding ack_i messages from a majority of k NameNodes, or (b) to be contained in a consensual state of a failover.

The condition (a) implies that a majority of NameNodes send a corresponding ack_i message, and this means a majority of NameNodes contain the Req_i in its state. In this situation, even if the Primary NameNode fails, Req_i is contained in the consensual state because at least one Req_i is considered in the consensus since the majority of NameNodes record its own state.

The condition (b) means that even if Req_i is unfixed, it can be contained in the consensual state on a failover. To restart HDFS service, the new Primary NameNode sends $Consensus()$ messages with the consensual state (including Req_i) to all Backup NameNodes and waits *ready* messages are received from a majority of the NameNodes. This guarantees that a majority of the NameNodes apply Req_i to their own states, and this causes the same configuration as the condition (a). □

The following Lemma 3 is induced from Lemma 2.

Lemma 3. *A transactions Tx_i with $Res_i = Reply$ is included in the sequential history π constructed by STEPs 1 and 2.*

Proof of Lemma 3. Since Res_i is $Reply$, there is a $Send_i$ event in H_S. The $Send_i$ event occurs when ack_i messages are received from a majority of NameNodes. This means that Req_i is in σ_S as the fixed state.

As a result, Req_i is never deleted from Lemma 2. Therefore, the corresponding transaction Tx_i is not deleted by STEP 2.

If there is no corresponding Get_i, Res_i cannot be $Reply$. Thus, Get_i is not removed by STEP 1, either. □

Lemma 3 guarantees that each Req_i, whose corresponding Res_i is $Reply$, is contained in σ_S. It follows that if Req_i is removed from σ_S only when the corresponding Res_i is $TimeOut$.

A Get_i event adds a new unfixed Req_i to the state of the Primary NameNode. Each unfixed Req_i becomes eventually: (a) to be changed to fixed by receiving ack_i messages from a majority of the NameNodes, (b) to be contained in a consensual state and becomes fixed, or (c) not to be contained in a consensual state and be removed from σ_S.

In the cases of (a) and (b), Req_i becomes fixed. This means that Req_i is applied to σ_S eventually. In the case of

(c), Req_i can be excluded from the consensual state. This implies that Req_i is never applied to σ_S.

Now we can say that the sequential history π constructed by STEPs 1 and 2 is legal from Lemmas 1 to 3. Thus Condition 1 of Definition 7 is satisfied. And Condition 2 of Definition 7 is trivial from Lemma 3.

Now we show that π constructed by STEPs 1 and 2 does not violate Conditions 3 and 4 of Definition 7.

Lemma 4 shows us that the condition 3 is not violated.

Lemma 4. *If Res_a precedes Inv_b in H_G and Res_a is Reply, then Get_a precedes Get_b in H_S.*

Proof of Lemma 4. Consider a transaction $Tx_a = (Inv_a, Res_a = Reply)$ in H_G. Get_a and $Send_a$ surely appear in H_S because Res_a is $Reply$. And the order of these events is $Get_a \prec Send_a \prec Res_a$.

Let Inv_b be an invocation following Res_a ($Res_a \prec Inv_b$) in H_G. Certainly Get_b appears in H_S. (If not, Tx_b is removed in STEP 1, thus this cannot violate Condition 3 of Definition 7.)

From the causal relations introduced above, the order $Get_a \prec Send_a \prec Res_a \prec Inv_b \prec Get_b$ holds. Therefore, Get_a precedes Get_b. □

Lemma 4 guarantees that if Res_a precedes Inv_b in H_G, Get_a is applied before Get_b. This shows that Condition 3 of Definition 7 is never violated because all Inv events in the sequential history π constructed by STEPs 1 and 2 appear in the same order as that of all Get events in H_S.

Condition 4 is concerning about all events occurred on a single client Clt_i. Consider two events, $Res_a \prec Inv_b$, in $H_G^{(Clt_i)}$. If Res_a is $Reply$, Res_a is applied before Inv_b by Lemma 4. Therefore, we consider only the case that Res_a is $TimeOut$. This cannot guarantee that $Send_a$ appears in H_S, thus we cannot use Lemma 4.

Lemma 5 guarantees that Condition 4 is not violated.

Lemma 5. *If Res_a precedes Inv_b in $H_G^{(Clt_i)}$, Get_a precedes Get_b in H_S.*

Proof of Lemma 5. (Proof of contradiction) If Get_a and Get_b occur on the same NameNode, Lemma 5 trivially holds because of FIFO property. Thus we consider the case that Get_a and Get_b occur on different NameNodes, NN_a and NN_b respectively. This implies that a failover is operated between the two requests (Inv_a and Inv_b) which changes a Primary NameNode from NN_a to NN_b, because the client Clt_i sent Inv_a to NN_a before sending Inv_b to NN_b. Thus, $Resync(NN_a)$ precedes $Resync(NN_b)$. If NN_a is the initial Primary NameNode, $Resync(NN_a)$ does not exist, thus we assume $Resync(NN_a)$ is sent on initialization of the system to simplify.

Assume that Get_b on NN_b precedes Get_a on NN_a. This means Get_b precedes Get_a in H_S. To apply Get_b, $Resync(NN_b)$ is required before Get_b. Thus we can determined the order of these events as follows: $Resync(NN_a) \prec Resync(NN_b) \prec Get_b \prec Get_a$. And Req_a is never applied to σ_S because the Primary NameNode is already changed to NN_b.

In order that both of Req_a and Req_b are applied to σ_S, the order of Get_a and Get_b may become as follow: $Resync(NN_a) \prec Get_a \prec Resync(NN_b) \prec Get_b$. However, this order violates our assumption. □

In this Section, we show how to construct the legal sequential history π by applying STEPs 1 and 2 to H_G. This π does not violate Conditions 2 (by Lemma 3), 3 (by Lemma 4), and 4 (by Lemma 5). As a result, the global history created from any execution of our system satisfies all conditions of weak-linearizability, and finally Theorem 1 is proved.

6. Experimental Evaluations

Our proposed system guarantees the namespace's consistency regardless some failures. However to implement this fault-tolerance, our system requires some synchronizations. This may cause system overhead which may decrease the performance of HDFS, therefore, we implement our system to experimentally, and evaluate the overhead. Moreover, we compare our system's overhead with QJM's. Finally, we evaluate the effect of the load balancing briefly.

In our system, the Primary NameNode has to communicate with $k - 1$ Backup NameNodes to synchronize for each request. If the frequency of the requests increases, the synchronization overhead also increases. In our experiments, we evaluate our proposed system with various request frequencies.

6.1 Experimental Environments

We use 36 commodity servers for the experiments. Table 1 shows the specification of each server.

Note that we implement our system based on Hadoop 1.0.3, and evaluate the overhead with comparing to Hadoop 1.0.3. However, QJM is not supported on this version of Hadoop, therefore we use Hadoop 2.2.0 in the evaluations of QJM.

We fix the number of DataNodes to 20 over all experiments. In the original Hadoop, just one NameNode is added. QJM has one Active NameNode, one Standby NameNode, and many JournalNodes (we change the number of JournalNodes during experiments). Our proposed system has multiple Primary NameNodes for several fragments. However to simplify the experiments, only one Primary NameNode is implemented in our experiments. In order to process requests, communications among Primary

Table 1 Specification of each server.

Number of Servers	36
CPU	Intel Core i3 2100 (3.1 GHz)
RAM	DDR3 4 GB
HDD	S-ATAII 500 GB
OS	CentOS 5.7
Hadoop	HDFS 1.0.3 or HDFS 2.2.0 (for QJM)

NameNodes are not required, thus this configuration has no influence to evaluations of the synchronization overheads.

HDFS (including our proposed system) supposes a lot of clients. Therefore we implement a client application, which creates lots of client threads concurrently. Each client thread sends a request to the (Primary) NameNode and disappears when the request is completed.

6.2 Synchronization Overheads

In this Section, we evaluate the overheads of synchronization among a Primary NameNode and Backup NameNodes and compare it with the original HDFS (without any synchronizations).

6.2.1 Overhead on Low Load

A client periodically sends a request to the (Primary) NameNode, and we evaluate the time until the response comes back. A client sends 300 requests in total, and we calculate the average time to process requests. And we repeat this evaluation (300 requests) 10 times for accuracy. In HDFS (including our proposed system), the experimental results vary widely on each experiment, especially when the system load is high. This implies that some results can be totally different from the other results even on the same condition. These unreliable results may cause the big change of the average of experimental results. To get the high reliable results, we calculate the average of the experimental result excepting the maximum and the minimum results.

A client sends a request every 100 ms (10 requests per second). This frequency is very low and uninfluential to the system performance.

Figure 8 shows the difference of the average process (response) time between the original HDFS and our proposed system. We change the number of Backup NameNodes from 3 to 13.

The original HDFS without any synchronization can process the request in about 7 ms. However, our proposed system requires about 22 ms, because of the synchronization overhead.

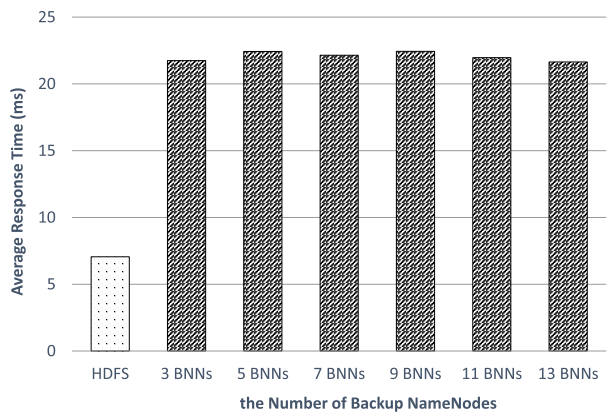


Fig. 8 Overhead on low load.

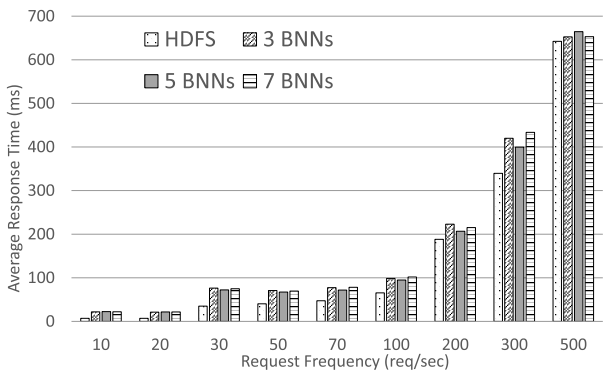
However, even when the number of Backup NameNodes is increased to improve fault-tolerance, the synchronization overhead is approximately same. Our synchronization protocol is majority-based, thus the Primary NameNode does not need to wait for all Backup NameNode's *ack*. This property makes waiting time for synchronizing stable.

6.2.2 Overhead on Heavy Load

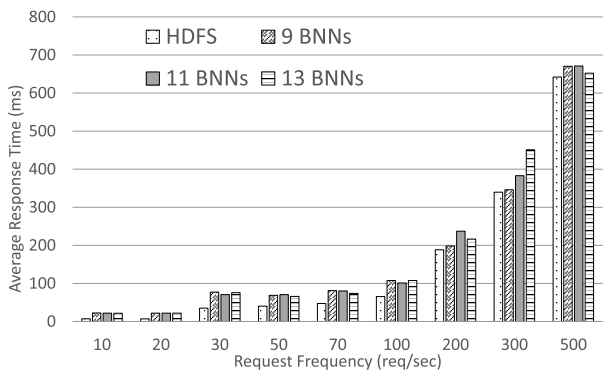
Now we change the frequency of the sending requests to evaluate average response time on various load environments.

Figure 9 describes the average response time on various request frequencies. X-axis shows the frequency of the requests, and Y-axis represents the average response time. Figure 9 (a) shows the result on 3, 5, and 7 Backup NameNodes compared with the original HDFS, and Fig. 9 (b) shows the result on 9, 11, and 13 Backup NameNodes.

As we discussed in the previous section, the orignal HDFS's response time is shorter than our proposed system's when the system load is low. However, if the frequency of the requests exceeds 200 requests per second, the difference becomes smaller. In more detail, we found the average overhead of the synchronization is about 15 ms in the case of low



(a)



(b)

Fig. 9 Overhead on heavy load.

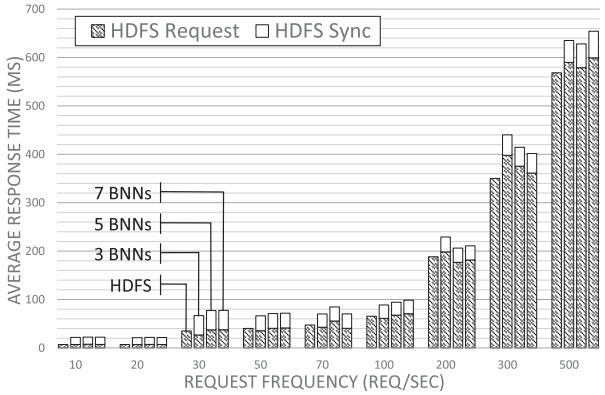


Fig. 10 Ratio of synchronization overhead.

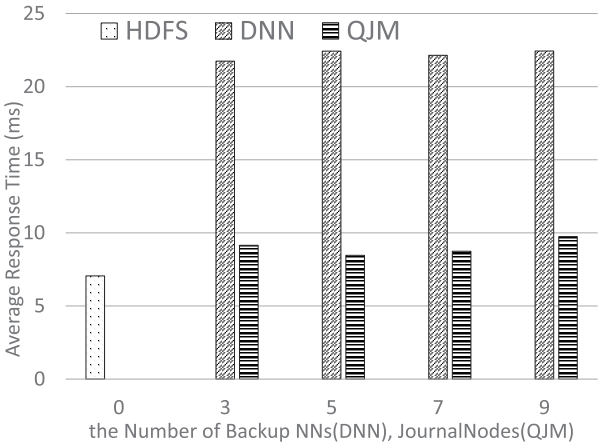


Fig. 11 Comparing with QJM on low load.

load. This overhead is relatively high because each request can be processed in about 7 ms. However, as the system load increases, the synchronization overhead gradually increases up to about 50 ms, about 3.3 times of the low load case. This increased amount of the overhead is relatively small because the HDFS overhead increases from about 7 ms to about 580 ms. As a result, the proportion of synchronization overhead is about 68% on the low load case, and becomes less than 8% on the heavy load case. And also in these cases, our system's response times are almost same regardless the number of Backup NameNodes.

From the result of Sect. 6.2, we can find that our system has synchronization overhead. However this overhead is small enough and has no effect on the system with high processing load.

To evaluate the actual synchronization overhead, we conduct the additional experiments to evaluates the request processing time and the synchronization time separately. Figure 10 represents the ratio of the synchronization overhead in the case of 3, 5, and 7 Backup NameNodes. Note that no overhead occurred on the original HDFS. The request processing time increases drastically when the request frequency increases. However, the synchronization time hardly increases even if the system's processing load is heavy. From this experiment, we can find again that the synchronization overhead of our proposed system exerts only little effect especially in heavy load environments.

6.3 Overhead Compared with QJM

Synchronization is also required in QJM on Hadoop 2.2. In this Section, we compare the synchronization overhead between our system and QJM. We set the number of the nodes which require synchronization (Backup NameNodes in our system, and JournalNodes in QJM) to be the same.

6.3.1 Overhead on Low Load

The same as Sect. 6.2.1, we fix the request frequency to 10 requests per second, and a client sends 300 requests. We repeat this 10 times and calculate the average response time.

Figure 11 represents the average response time on our system and QJM when the number of Backup NameNodes (or JournalNodes in QJM) is increased. In QJM, significantly different from our proposed system, it can process the request in about 9 ms, this result is only 2 ms slower than the original HDFS. The optimization of the synchronization process can be cited as a possible cause. QJM also hardly changes the average response time if the number of JournalNodes is increased.

6.3.2 Overhead on Heavy Load

In this Section, we change the request frequency, and measure the average response time of our system and QJM.

Figure 12 shows the average response time when the frequency of the requests is changed. The same as the previous section, X-axis represents the request frequency, and Y-axis shows the average response time. Figure 12 (a) shows the result on 3 and 5 Backup NameNodes, and Fig. 12 (b) shows the result on 7 and 9 Backup NameNodes (JournalNodes in the case of QJM).

In the result in the Sect. 6.3.1, QJM has lower synchronization overhead than our system, but we can find the inverse result in this experiment. QJM's average response time increases faster than our system's when the frequency of the requests increases. QJM has faster reponse time than our system under 30 requests per second, however becomes slower over 50 requests per second. QJM requires some communications with Standby NameNode to keep their states identical, this makes a response time slower on heavy load environments.

6.4 Effect of Load Balancing

In this Section, we evaluate the effect of the load balancing by some experiments. Now we implement a several number of Primary NameNodes to consider the case that the namespace is partitioned into *m* fragments.

In this experiment, a client sends *mkdir* (make a directory) request periodically, and the target directory name

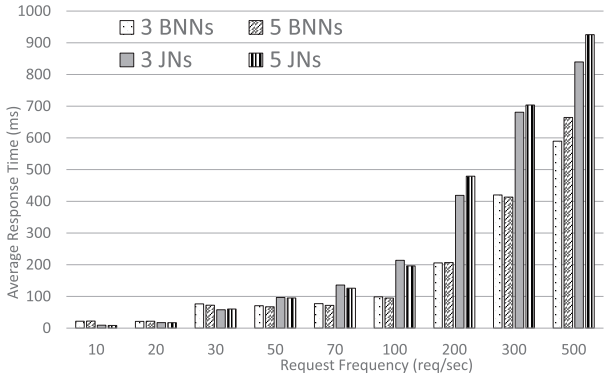


Fig. 12 Comparing with QJM on heavy load.

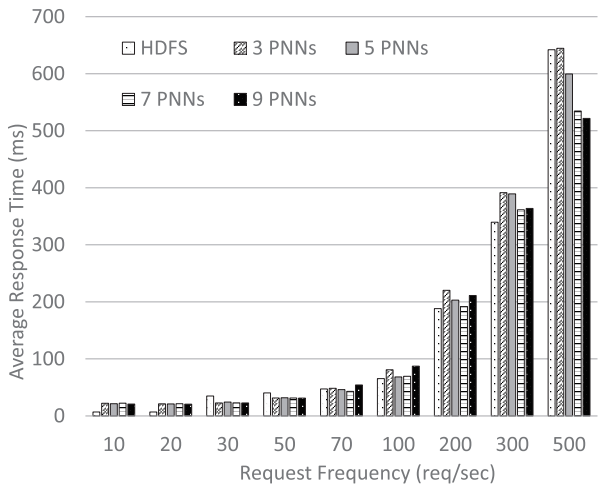


Fig. 13 Effect of load balancing.

of *mkdir* is created randomly. And the target Primary NameNode is determined by this target directory's name. We fix the number of Backup NameNodes, however they also can be a Primary NameNode of other fragments. This implies that a NameNode can be a Primary NameNode for some fragments and a Backup NameNode for other fragments concurrently.

Figure 13 represents the average response time when we change the request frequency from 10 requests per second to 500. To help to clarify the effect of the load balancing, we compare it with the original HDFS. When the frequency is low (low processing load), we can not find the effect of the load balancing because the overhead is higher in this environment. However, in the case of the highest frequency (500 requests per second), our system with 9 Primary NameNodes (this means 9 fragments exist) has the shortest response time.

We can conclude that the response time cannot be decreased linearly, even when all requests sent by clients are processed concurrently by a several Primary NameNodes, from following two reasons.

At first, we set the target directory of *mkdir* request randomly, this means that we never apply some techniques to improve the effect of the load balancing. This cannot guarantee that the frequencies of the requests received by Primary NameNodes are different among Primary NameNodes.

Secondly, as we mentioned above, some Primary NameNode can be also a Backup NameNode of a different fragment. This implies that it is required to process the request and synchronization with a different Primary NameNode concurrently.

However, we can find the clear effect of the load balancing when the processing load is heavy. And this effect becomes more effective, when the number of the fragments m is large.

6.5 Performance of MapReduce Task

As we introduce in the Sect. 1, Hadoop consists of two main components, HDFS and MapReduce. Hadoop can execute distributed processing using MapReduce framework. In this Section, we focused on the evaluation of our system's overhead comparing with HDFS and related work (QJM). Certainly, these results show only the performance of the file system (HDFS).

However, we hardly evaluate the performace of Hadoop (MapReduce) task because it is influenced a great deal by a namespace partitioning rule. If a Hadoop task can be executed on the single fragment, our system can execute the task with the same processing time, because our system can operate in exactly the same manner. If a Hadoop task needs to access two or more fragments concurrently, our system may process the task separately on each fragment. After that, a specific process is required to merge the separated results. To obtain good performance, the effective namespace partitioning is required. We consider that confirming the partitioning rule is a future work.

7. Summary and Future Works

In this paper, we proposed a new architecture of HDFS, which partitions the namespace into m fragments and manages them in a distributed manner using m Primary NameNodes. Our proposed system consists of m Primary

NameNodes and many Backup NameNodes (depending on k) to guarantee the fault-tolerance. Actually we resolved the SPOF problem of HDFS and the namespace limitation problem, and also got the effective load balancing. We proved the consistency of the namespace regardless some failures in the system. Our proposed system is majority-based, and this can keep the namespace's state consistent. Finally, we evaluated our system's synchronization overheads and the effect of the load balancing compared with the original HDFS and QJM.

To make our system completely practical, we should enhance some capabilities. First of all, to improve the performance, the effective partition policy of the namespace is required. In our experiments, we had partitioned a namespace based on directory names. However this may cause high processing load on some specific NameNodes. We can also consider the dynamic paritioning to resolve the above problem. This partioning causes some effects to the performace of the Hadoop task as we explained in Sect. 6.5. Moreover, some integrated API (Application Programming Interface) to control our system is required. For example, a method which browses the entire namespace is required for clients.

Acknowledgements

The authors would like to thank Prof. H. Kakugawa and Prof. F. Ooshita at Osaka University for meaningful discussions about improving this paper. This work is supported in part by Grant-in-Aid for Scientific Research ((B) 26280022) and ((B) 24650012) of JSPS.

References

[1] J. Gantz and D. Reinsel, "The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East," IDC iView, 2012.
[2] Apache Hadoop, The Apache Software Foundation, http://hadoop.apache.org/
[3] K. Shvachko, "Warm HA NameNode going Hot," Apache Hadoop Issues, HDFS-2064, 2011.
[4] Big data, http://en.wikipedia.org/wiki/Big_data
[5] S. Ghemawat, H. Gobioff, and S-T. Leung, "The Google file system," Proceedings of ACM Symposium on Operating Systems Principles, pp.29–43, 2003.
[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp.137–149, 2004.
[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," Proc. IEEE the 26th Symposium on Mass Storage Systems and Technologies (MSST), pp.1–10, 2010.
[8] Apache ZooKeeper, The Apache Software Foundation, http://zookeeper.apache.org/
[9] F.P. Junqueira and B.C. Reed, "The life and times of a zookeeper," Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC), p.4, 2009.
[10] T. White, Hadoop: The Definitive Guide, 3rd edition, O'Reilly Media, Yahoo! Press, 2012.
[11] HDFS High Availability, The Apache Software Foundation, http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithNFS.html
[12] Todd Lipcon, "Quorum-based protocol for reading and writing edit logs," Hadoop HDFS Issues, HDFS-3077, 2012.
[13] D. Molkov, "Hot Standby for NameNode," Apache Hadoop HDFS Issues, HDFS-976, 2013.
[14] A. Ryan, "Under the Hood: Hadoop Distributed Filesystem reliability with Namenode and Avatarnode," Facebook, 2012. https://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920
[15] Giraffa File System, A distirbuted highly available file System, https://code.google.com/a/apache-extras.org/p/giraffa/wiki/Introduction
[16] Apache HBASE, The Apache Software Foundation, http://hbase.apache.org/
[17] L. Lamport, "Paxos made simple," ACM SIGACT News, vol.32, no.4, pp.18–25, 2001.
[18] K. Shvachko, "HDFS scalability: The limits to growth," ;login:, vol.35, no.2, pp.6–16, 2010.
[19] K. Shvachko, "Scalability of the Hadoop Distributed File System," Hadoop Blog in Yahoo! Developer Network, 2010.
[20] K. Shvachko, "Name-node memory size estimates and optimization proposal," Apache Hadoop Common Issues, HADOOP-1687, 2007.
[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable: A distributed storage system for structured data," ACM Transactions on Computer Systems (TOCS), vol.26, no.2, article 4, pp.1–26, 2008.
[22] H. Attiya and J. Welch, Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd Edition, Wiley, 2004.
[23] M.P. Herlihy and J.M. Wing, "Linearizability: A correctness condition for concurrent objects," ACM Transactions on Programming Languages and Systems (TOPLAS), vol.12, no.3, pp.463–492, 1990.
[24] M. Raynal, Distributed Algorithms for Message-Passing Systems, Springer, 2013.
[25] G. Kola, T. Kosar, and M. Livny, "Faults in large distributed systems and what we can do about them," Proceedings of 11th International Euro-Par Conference, pp.442–453, 2005.

Yonghwan Kim is a Ph.D candidate at Graduate School of Information Science and Technology in Osaka University, in Osaka, Japan. He received the B.E. double-degree in Electronics and Informatics from Soongsil University in Seoul, Korea, in 2009. And he received the M.E. degree in Informatics from Osaka University in 2011. He has lots of developing experience as a member of Samsung Software Membership at Samsung Electronic Company, Korea. The areas of research interests include distributed computing, fault tolerance and distributed file systems.

Tadashi Araragi received the B.E. and M.E. degrees in Mathematics from Tokyo University, in 1985 and 1987, respectively, and D.E. degree in Informatics from Kyoto University in 2006. He joined Communications and Information Processing Labs. of NTT (NIPPON TELEGRAPH AND TELEPHONE CORPORATION), Yokosuka, Japan in 1987 and also worked as a Senior Research Scientist at NTT Communication Science Laboratories until 2014. He is currently working at Proassist, Ltd, Osaka, Japan. His research interest includes formal verification, analysis of security protocols and fault tolerance of distributed systems. Dr. Araragi is a member of IEICE and IEEE Society.

Junya Nakamura received the B.E. and M.E. degrees from the Department of Knowledge-based Information Engineering, Toyohashi University of Technology in 2006 and 2008 respectively, and D.E. degree in information science from Graduate School of Information Science and Technology, Osaka University in 2014. He is currently a project assistant professor of Information and Media Center, Toyohashi University. His research interests include distributed algorithms and fault tolerance of distributed systems.

Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993-1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, IEICE and IPSJ.