International Journal of Grid and Distributed Computing
Vol. 9, No. 5 (2016), pp.175-190
http://dx.doi.org/10.14257/ijgdc.2016.9.5.16

An Efficient Replicated System for the Metadata of HDFS

Zhanye Wang1, Tao Xu1, Dongsheng Wang2

[1]*Department of Computer Science and Technology*
[2]*Research Institute of Information Technology*
Tsinghua University
Beijing, 100084 P.R. China
{wangzhanye10, t-xu10}@mails.tsinghua.edu.cn, wds@tsinghua.edu.cn

Abstract

Hadoop HDFS is an open source project from Apache Software Foundation for scalable, distributed computing and data storage. HDFS has become a critical component in today's cloud computing environment and a wide range of applications built on top of it. However, the initial design of HDFS has introduced a single-point-of-failure, since HDFS contains only one active namenode, if this namenode experiences software or hardware failures, the whole HDFS cluster is unusable, this is a reason why people are reluctant to deploy HDFS for an application whose requirement is high availability. In this paper, we present a solution to enable the high availability for HDFS's namenode through efficient metadata replication. Our solution has 3 major advantages than existing ones: We utilize multiple active namenodes, instead of one, to build a cluster to serve requests of metadata simultaneously; We implement a pub/sub system to handle the metadata replication process across these active namonodes efficiently; We also propose a novel replication algorithm to deal with the network delay when the namonodes are deployed in different areas. Based on the solution we build a prototype called NCluster and integrate it with HDFS. We evaluate NCluster to exhibit its feasibility and effectiveness. The experimental results show that our solution performs well with low replication cost, good throughput and scalability.

Keywords*: HDFS; namenode; metadata; availability; replication; NCluster*

1. Introduction

Apache Hadoop[1] is an open source Java project governed by the Apache Software Foundation(ASF), and is being built and used by a global community of contributors. Hadoop is a reliable, scalable, distributed computing platform, clients usually use simple programming model to run distributed computing tasks like MapReduce[2] over a massive amount of data sets on Hadoop platform. It is designed to scale up from single server to thousands of machines, each offering local computation and storage. There are many sub open source projects belongs to Hadoop, including HBASE[3], Zookeeper[4] , *.etc*. Maybe the most famous one of these projects is HDFS[5], a distributed file system can be scalable to support thousands of commodity machines, HDFS is the primary distributed storage system used by Hadoop applications . Industry leading companies, such as Facebook, Twitter and YAHOO!, use HDFS as their basic distributed storage environment[6].

The HDFS cluster consists of hundreds even thousands of machines, in this paper we call these machines nodes. The design of HDFS is fully inspired by Google File System (GFS)[7]. Both of HDFS and GFS are master/slave architecture. Each HDFS cluster has one master node, called namenode, which manages the metadata information, including distributed file system namespace, file descriptions, file-data block mappings, data block allocations, access regulations and so on. HDFS clients should talk to the namenode

ISSN: 2005-4262 IJGDC
Copyright ⓒ 2016 SERSC

firstly whenever they wish to access a file in HDFS. Note that there could be more than one namenodes in each HDFS, but only one of them is *active*, here *active* means it can serve requests from HDFS clients, in contrast to *inactive* namenode, which merely play a role of warm or cold backup. Within HDFS cluster, in addition to namenode, there are a huge number of datanodes, which are responsible for storing the actual data blocks. Rather than rely on hardware to deliver high-availability, the HDFS itself is designed to detect and handle failures at the application layer, it uses multiple data replicas across the cluster. In default setting, every data block has 3 replicas, which resident in 3 different datanodes respectively, replicas are updated synchronously in order to provide strong consistency. Even though each of datanodes may be prone to failures, but the data loss of one or few datanodes does not impact the HDFS cluster at all, HDFS clients could fetch the data from other datanodes with same replica.

Unfortunately, HDFS has quite a little high availability support on namenode yet, existing solutions are far from satisfaction. One major difference between HDFS and GFS is that, the initial design of HDFS has only one active metadata server as mentioned earlier, while GFS has 3 or 5 for the reason of high availability, GFS uses Chubby Lock[8] to maintain metadata files across different metadata servers.

Furthermore, in HDFS any time when the namenode meets unexpected errors or system failures, the whole HDFS cluster is totally out of work because the metadata is inaccessible. Apparently this is a single point of failure and is one of the reasons why people are reluctant to deploy HDFS for an application whose uptime requirement is 24x7. Namenode may fail to response request due to many reasons, from unexpected power off or hardware crash, to software upgrades or malicious invasions.

At present, there are a number of solutions for the prevention of single point of failure of HDFS namenode, such as Secondary namenode, Avatar namenode, *.etc*. While the drawbacks of these solutions are organized as follows:

- There is only one active namenode in each HDFS cluster, the rest of namenodes only play a role of backup server, since these inactive namenodes cannot serve clients' requests, it is kind of a waste of hardware resources.
- During the process of metadata backup, active namenode firstly persists the updates of metadata on its local disk as files, then replicates these files to the inactive namenodes, finally inactive namenodes load the files of metadata updates in their own main memory. Clearly we can see that the backup process causes heavy disk I/O and limits the speed of metadata replication and the overall performance of HDFS.
- Due to high network latency, the backup process is time-consuming if the backup server is deployed in the remote area for the reason of disaster tolerance.

In this paper, we present a solution to improve the high availability for the namenode in HDFS . We build a cluster contains a small number of namenodes(typically 3 to 9),this cluster uses single writer multiple readers strategy. One namenode called primary, it handles all the metadata update requests(create, mkdir, delete, *.etc*) and a portion of read requests from HDFS clients, the rest of namenodes are called hot standbys, they are all read only. During the uptime, the primary replicates all the metadata updates to the hot standby namenodes continuously, we implement a pub/sub system to handle these replication process, the pub/sub system do not persist any metadata on disk of primary or hot standbys to ensure high efficiency. When the primary namenode crashes, the time of failover will be finished in a few seconds. Moreover, the hot standbys could be deployed in remote data centers for reason of disaster tolerance, we propose a novel replication algorithm to deal with the high network latency.

 Copyright ⓒ 2016 SERSC

2. Background and Related Work

2.1. Metadata of HDFS

HDFS is a distributed file system currently being used in situations where massive amounts of data need to be processed. It offers a way to store large files across thousands of physical or virtual machines. It is the building block of Apache Map-Reduce framework. Like Linux VFS[10],the metadata of HDFS is also managed by the data structures called inode, inode contains information like file or directory descriptions (path, permissions, leases, *etc*.), file-data block mappings , data block allocations and so on.

When HDFS is running, namenode receives updates of metadata from HDFS clients continuously, datanodes also send block report to namenode periodically to tell it whether there are corrupted data blocks or not. Before namenode applying these updates to the inode structures in main memory, namenode also log each updates into a file called Editlog, for it can construct the metadata by replaying the updates in Editlog.

In HDFS, the namenode is the single point of failure, if namenode experiences software or hardware failure, the whole cluster will go down. Moreover, namenode also has become the performance bottleneck of the HDFS cluster since there is only one active namenode exists in cluster. Many previous works have discussed how to improve the availability and performance of HDFS's namenode. We will make some brief introductions of these solutions in the following paragraphs.

2.2. Metadata Availability

2.2.1 Secondary Namenode: The HDFS itself provides an high availability solution for namenode called Secondary namenode[11]: as shown in Figure 1, in the HDFS cluster there are two namenodes, namely Primary namenode and Secondary namenode. Secondary namenode is not active, since it does not respond to any read or update request of the metadata, all it needs to do is periodically fetch EditLog from primary namenode. When failover occurs, secondary namenode read metadata from EditLog from its local disk, then take the role of primary namenode and respond to requests from HDFS clients.

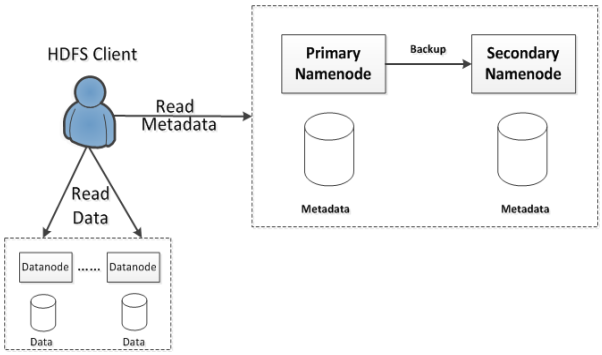

Figure 1. Secondery Namenode

Unfortunately, the failover process of this solution above is time-consuming because the Secondary or the backup namenode is not a real-time mirror to primary namenode, they backup the metadata periodically .Statistics from Facebook indicate[12] that for a HDFS contains thousands of datanodes, the recovery of Secondary namenode spends tens of minutes. While the solution we present is that,

Copyright ⓒ 2016 SERSC

all the metadata in each namenodes are identical in real time, so the time of failover will be finished in very soon.

2.2.2 Avatar Namenode

To shorten the process of failover, Facebook presented a solution called Avartor namenode[13], similar to Secondary namenode, the HDFS contains two namenodes, one called active namenode, the other one called standby namenode, the active is used to serve read and write requests from HDFS clients, after failover occurs standby becomes the new active namenode, the major difference from Secondary namenode is that the active and standby share metadata files by using NFS, after the active namenode store the Editlog into a shared storage device, while standby namenode soon will read the updates of metadata in real time.

However, a shared storage device itself becomes another single point of failure to the system. The other disadvantage is that, the backup is not active either, it is merely warm backup. In our solution all the namenodes are active, and the experimental results shows that our solution has excellent system throughput and scalability.

2.2.3 Integration with Zookeeper

ZooKeeper is an open source project from the Apache Software Foundation, providing a distributed lock service and synchronization service for large distributed systems. As shown in Figure 2, some deploys Zookeeper on multiple namenode to synchronize metadata of HDFS[14]. All of these namenode are active, one namenode called leader handles all the write requests, it also uses Zookeeper to replicate the updates of metadata to the rest of namenodes for data consistency.

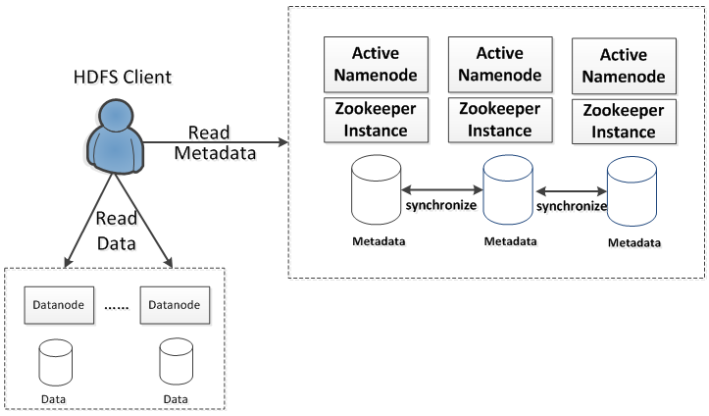

Figure 2. Schema of Integration with Zookeeper

However, the process of metadata replication is inefficient. Because frequent synchronization of metadata files across namenodes results in a large amount of disk I/O, which limits the speed of replication. One can use SSD instead of hard disk drives to accelerate the metadata replication, since SSD's I/O is much faster than HDD's. But SSD has its own shortcomings, the most notable one is its limited lifetime span when serving massive write requests. On the contrary, we implement a pub/sub system to handle these replication process, the pub/sub system do not persist any metadata on disk of primary namenode or hot standby namenodes, which enables high efficiency.

 Copyright ⓒ 2016 SERSC

2.3. Replication

Basically, our prototype uses replication techniques to enhance the availability and throughput of read operation of namenode. Primary is in charge of the metadata replication in order to make sure every namenode in NCluster has the identical metadata.

The updates of any replica introduce the issue of data consistency. Some replicated systems adopt strong consistency, such as [5][20], briefly speaking strong consistency requires the replicated system acts like a single server that serializes all operations, the read operation will never fetch stale results from any replica. To accomplish this goal, the replicated systems have to update all the replica synchronously, which ensure that every replica receives the same sequence of updates and yield the same value. Almost all the replicated systems with strong consistency resident within the local-area network, in which the network delay remains quite little(tens of millisecond), so the replication process is relatively fast even though it have to propagate the update to every replica.

For the reason of disaster tolerance and low latency access, we hope our prototype could replicates the metadata into different area (data center). Clearly the strong consistency does not work well if the replicas are geo-replicated, for the network delay is much higher than the local area, and network congestion occurs often which could push the time delay to several seconds. The geo-replicated systems[19] usually only guarantees the eventual consistency. In these systems the updates propagate to each replica asynchronously which shortens the time of replication. Unfortunately, systems that embrace eventual consistency have limitations because replicas may diverge in the short term as long as the divergence is eventually repaired. During this short term some replicas may exhibit stale status, and if some application read this stale result they would produce a wrong value.

3. System Design

In this section, we depict the system design of NCluster. We build a cluster contains a small number of namenodes (typically 3 to 5) to manage the metadata requests, the cluster uses single writer multiple readers strategy. One namenode called primary namenode, it handles all the write requests and a portion of read requests from HDFS clients, the rest of namenodes are called hot standby namenodes, they are all read only, which means they could only serve the read requests.

Meanwhile, since we have multiple namenodes, for the reason of consistency the primary namenode is also responsible for propagating the updates of the metadata to the hot standby namenode. We implement a pub/sub system to deal with the metadata replications, we find the process is quite efficient because the pub/sub mechanism requires no disk I/O. In the following paragraphs we will describe the details of NCluster.

Copyright ⓒ 2016 SERSC

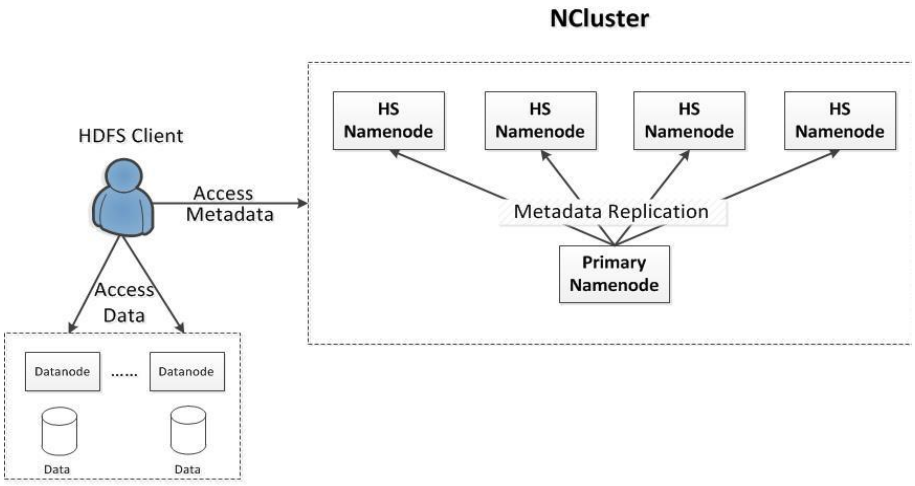

Figure 3. The Architecture of NCluster

3.1 Architecture

Figure 3 exhibits the architecture of NCluster. NCluster consists of multiple namenodes, HDFS clients access metadata from NCluster, the update request will direct to primary namenode, the read request will spread evenly across all the namenodes, for the reason of data consistency, primary namenode is also responsible for metadata replication. Next we will focus on how to replicate metadata effectively and efficiently.

3.2 What to Replicate

When HDFS is running, the metadata is stored in the main memory of primary namenode for access efficiency, it is managed by the data structures called inode. One simple strategy is to replicate the entire inode structure when each time it has been modified, we did not apply this because inode contains a lot of information, but often only small portion of it would be changed in each update.

Instead of replicating the whole inode structure, when the inode structure is modified by an update operation, the primary namenode only replicate the type and the parameters of this update, take mkdir (to make a directory) operations as an example, the primary propagates the path of the directory, permissions and timestamp to all the hot standbys.

Before primary namenode log each updates into EditLog, it propagates the type of updates and their parameters to every hot standbys, then each hot standby re-executes the updates according to the information it has received from the primary, to make all the metadata in NCluster are identical.

3.3. How to Replicate

As described previously, one can utilize Zookeeper to build multiple active namenodes for HDFS. However, this approach could cause intensive disk I/O which limits the performance of replication seriously. We implement a pub/sub system based on open source project ActiveMQ[15] to deal with metadata replication efficiently in contrast to Zookeeper schema.

The general idea behind pub/sub model is the design pattern called Observer[16], one benefit of using this model is the ability to break down the applications into smaller, more loosely coupled modules. There are three roles in pub/sub model: publisher, subscriber and topics. Topics can be considered as logical channels between publisher and

 Copyright ⓒ 2016 SERSC

subscriber. The publisher can put messages into a certain topic, the subscriber subscribe that topic could get the message instantly.

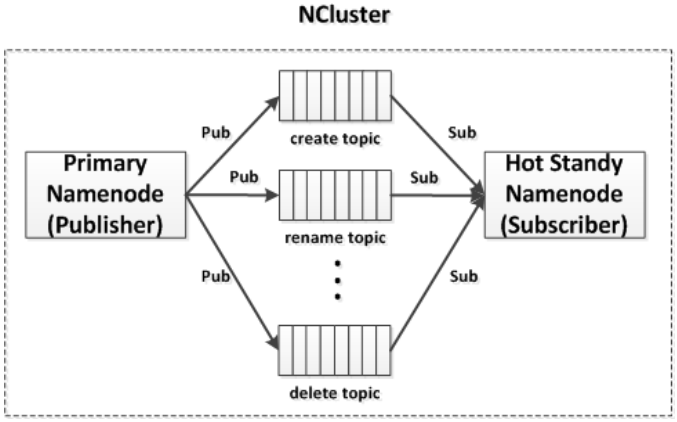

Figure 4. Pub/Sub System

3.4 Consistency

Since NCluster has multiple active namenodes, so it introduces the issue of data consistency. As all we know, it is very difficult to achieve both of strong consistency and low access latency in a distributed storage environment.

So in the design of NCluster, we provide two replication mode for HDFS clients in the local network environment, namely Sync and Async. Sync means that, after HDFS clients issue a write request to primary namenode, it is blocked until the metadata have been propagated to all the hot standby namenodes. If any of the hot standby does not receive the metadata, or it does not response the primary node over the certain time threshold (3s in our default setting), this write operation will be rollback. Obviously, Sync write follows the strong consistency[17] , read operation to any of the namenode will return the newest data.

NCluster also offer the Async mode to improve the performance and throughput. When NCluster is set with Async, HDFS clients issue a update request to primary namenode, it is only been blocked until receive the ack from primary namenode, so it is faster than Sync, but the Async only provides a weaker consistency called eventual consistency, it only guarantees that the read requests to primary namenode will get the newest metadata, the hot standbys may contains the inconsistent (stale) metadata. Since Async is designed for the local network environment, metadata in different namenode may diverge in very short term as long as the divergence is eventually repaired, the chance of reading an inconsistent metadata is low. In the next section we will propose an efficient replication algorithm called 2PR when the namenodes in NCluster are geo-distributed.

3.5 Replication Optimization

As mentioned previously, NCluster can deploy its namenodes at different areas. The reason we replicate the metadata of namenode across data centers or sites is to provide disaster tolerance, access locality, and read scalability. Note that in this paper we only focus on how to make namenodes geo-distributed, the corresponding issue about datanodes is beyond the scope of the paper, moreover, the issue of geo-distribution about datanodes has been discussed intensively in [19][20][21].

Both of Sync and Async are designed for local-area networks, these algorithms perform poorly when namenode are spread over remote sites, Sync prolong the waiting time of the HDFS client, and it is very likely to fetch the stale metadata if NCluster utilizes Aysnc because the divergence of metadata may exists for a long time. A solution

Copyright ⓒ 2016 SERSC

to this problem is to provision the cross-site links for peak usage but this solution could be expensive and wasteful.

In this section, we narrate our novel replication algorithm called 2PR(2 phase replication) for the situation when the namenodes in NCluster are geo-distributed, the essence of 2PR is to break each process of metadata replication into two phase, the first phase called timestamp replication, in this phase the primary namenode only replicates the timestamp information synchronously, after phase one the primary could inform the HDFS client that the metadata update is complete. So the client only have to wait a small amount of time contrasts to Sync since the size of timestamp information is small(*e.g.* tens of bytes), in the second phase the primary continue to replicate the metadata in the background and that does not block the HDFS client. The 2PR can ensure strong consistency by checking the timestamp from all the namenodes to find which namenode contains the newest metadata.

Settings: We consider NCluster is able to deploy it namenodes in multiple data centers or sites, where each data center are connected to each other by wide-area network links which have lower bandwidth and higher latencies than the links within a data center. Every namenodes share a synchronized clock, which are used as timestamp. This clock can be implemented with GPS sensors, atomic clock, radio signals, or protocols such as NTP, Spanner[21] has shown these are feasible even when a system are geo-distributed. At last, we do not consider Byzantine failures in the design of NCluster.

Goals: We must still continue to provide strong consistency when each process of metadata replication has been split. The difficulty here is that each update that is split in two parts may be interleaved with other updates, creating concurrency problems. To address such problems, the read protocol of 2PR includes some additional phases of communication and coordination to make sure every read operation from HDFS client can fetch the newest metadata from NCluster.

Details: We now describe the 2PR algorithm in more details. As shown in Figure 5, there are one primary and two hot standbys in NCluster, timeline grows from top to bottom. At the very beginning, HDFS client sent a metadata update request to the primary namenode, the primary's job is to propagate the data of this update to the other 2 hot standbys efficiently, the data includes the type of the update (create, delete, *.etc*) and the its parameters as we discussed earlier, the HDFS client has been blocked until the replication process is complete. To replicate the update, the primary acquires a global timestamp as the sequence number of the update and starts the 2PR. From the Figure 5 we can see that in the first phase, the primary namenode only replicate the timestamp information synchronously, after primary receives from ack of timestamp from all the hot standbys it informs the HDFS client that the metadata update is complete and the client is no longer blocked. The size of timestamp and ack is only 8-16 bytes, so the phase 1 is not time-consuming even though the primary and hot standbys are geo-distributed. That also means the blocked time of HDFS client is short and it is the reason why 2PR is much more efficiently than Sync. In the phase 2, primary starts to propagate the data of the update(type and parameters) asynchronously together with a same timestamp with phase1, the size of this data in phase 2 is usually large, typically tens of KBs, which is orders of magnitude larger phase 1.

 Copyright ⓒ 2016 SERSC

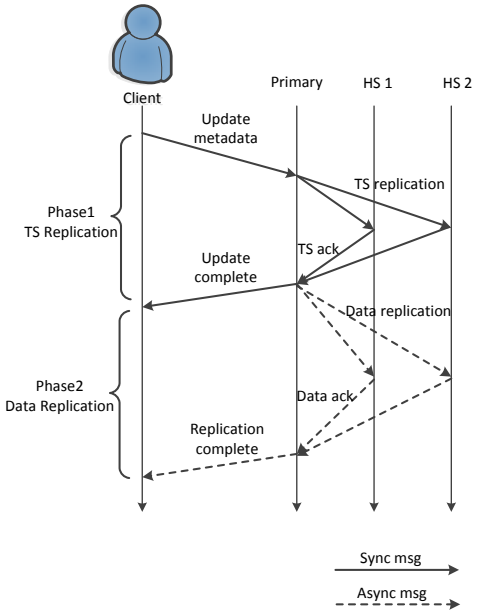

Figure 5. Detailed Process of 2PR

To read a consistent metadata, the client uses a small message to ask all the namenodes to send their *newest consistent timestamp*. Take the situation shown in Figure 6 as an example, a hot standby has received 3 updates of metadata from the primary through 2PR, the timestamps of these updates are 1, 2, 3 respectively. However, in this very special case the data of 3rd update came earlier than the 2nd update because the data replication(phase 2) of 2PR are processed asynchronously and it cannot guarantee the sequence of data received by the hot standby, maybe the data of 3rd updates is much smaller than 2nd one so it is received earlier. Clearly timestamp 2 is not consistent since it has not received the data yet, timestamp 3 is inconsistent either because the 2nd and 3rd update may has some causality relationship (create a file and immediately rename it), if hot standbys execute 3rd update before 2nd one it will goes to a wrong status, so the newest consistent timestamp is Figure 6 is 1.

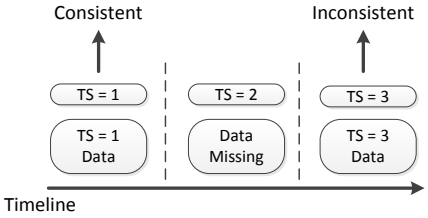

Figure 6. Consistent Timestamp

All the namenodes reply to client with a small message contains its newest consistent timestamp. Once the client has got all of the replies, it finds out the highest timestamp that it received. It then asks the corresponding namenode to send the data associated with highest timestamp. If there are more 1 namenodes with the highest timestamp, the client chooses the namenode it is close to. The reply is a message with data but, in the common case, a namenode in the local data center has the data, so this namenode responds quickly without remote communication. Thus, the client can read the consistent metadata without being affected by the congestion on the remote path.

Analysis: We now analyze the Async, Sync and 2PR these three replication algorithms. Table 1 summarizes the results. For the update operation, obviously the Async

Copyright ⓒ 2016 SERSC

is the fastest, for it only update the primary namenode, the 2PR is slightly slower than Async, for it has to update the primary and the all of the hot standbys with timestamp, the big advantage is that the 2PR can guarantee the strong consistency. Sync is much more time-consuming because it has to update the data to all the hot standbys which result in the client has to wait for a long period of time.

Algorithm	Update	Read	Consistency
Async	*U(primary, data)*	*R(local, data)*	*Eventual*
Sync	*U(primary, data) + U(hs, data)*	*R(local, data)*	*Strong*
2PR	*U(primary, data) + U(hs, ts)*	*R(hs, ts) +R(data)*	*Strong*

Table 1. Comparison of Async, Sync and 2PR

For read operation, both Async and Sync read the metadata from the namenode in local datacenter, but the Async may offer stale result. 2PR firstly send a small message to check which namenode contains the newest metadata, then sent the read request to namenode which sent this timestamp. So in the common case, the local namenode contains the data and the read request finished quickly. In the worst case, the HDFS client has to issue a read request to the remote data center.

3.6 Failover

Note that the failover occurs only when primary namenode is down. The hot standbys are simply responsible for read requests, so the system failure or crash of them does not impact the whole HDFS cluster.

Failover consists of 2 steps: leader election and IP address transition. The later step is relative simple, since the primary namenode of HDFS is accessed through IP address. When a certain hot standby is elected as the new primary namenode, it changes its IP address to the same as the old primary node, after that it is able to take over all communications with other namenodes and datanodes .

If NCluster only supports sync mode, new leader election will be easy hence all metadata across each namenodes are identical. When the primary namenode meets failures, it is feasible to choose any another hot standby namenode randomly as the new primary. Unfortunately, things gets more complicated in the situation of NCluster, since NCluster should also support Async and 2PR, the metadata exists among these namenodes may not be the same.

Here is our details of leader election, suppose NCluster contains N hot standby namenodes, initially we assign an increasing sequence of number to each of the hot standby, says 1 to N, when hot standbys believe the primary is out of work (they have not received the acknowledgement of their heartbeat from primary for a long time which exceeds a predefined threshold), the hot standby with the highest number broadcast a message to check which hot standby has the most recent metadata, then it sends this namenode's number to the all of hot standbys to make sure every hot standby can realize where to synchronize the newest metadata, when the synchronization process is done ,the hot standby with the highest number becomes the new primary node and takes charge of update request for metadata.

4. Implemention

In this section we will describe the implementation of NCluster briefly. The NCluser is based on HDFS source code 2.02, we modify approximately 1800 lines of Java code. These modifications are mainly at:

- Namenode.java: this source file handling the communication between primary namenode and the hot standby namenodes, including the heartbeat to detect

 Copyright ⓒ 2016 SERSC

whether any of them is healthy or not. It is also in charge of failover in case of primary namenode is down. Moreover, during initialization of hot standby, we start a thread to receive the metadata updates from the pub/sub system.

- Datanode.java: we modify this source file because we want to each datanode could register at every namenodes in NCluster, so it could receive request from all of them.
- ClientProtocol.java: we add a flag at the all the update request (create, rename, mkdir, delete, *.etc*) of this source file to identify whether this request will be running in Sync, Async or 2PR. the other one modification is adding the load balance to NCluster when it serves read requests, the strategy is simple: randomly spreading the read request across the NCluster.
- EditLog.java: originally, this file is responsible for logging each update of metadata issued by HDFS clients into EditLog. In the implementation of NCluster, the primary also puts all the updates it received and their parameters into pub/sub system for metadata replication before logging.

We implement the pub/sub system based on ActiveMQ 5.8.0 for its outstanding reliability and scalability. However, ActiveMQ utilizes Serializable interface provided by JDK to transfer Java Object over network. On the contrary, we use Writable interface provided by HDFS itself to achieve Object transfer because Writable interface is more efficient.

5. Evaluations

5.1. Experimental Setup

In this section we start to present the evaluation of the NCluster, including replication cost, Sync and Async throughput testing, scalability, the comparison to Zookeeper schemes and the replication optimization when NCluster is geo-distributed. All measurements were performed on HDFS clusters version 2.0.2. The HDFS integrates with a NCluster consists of 3 to 9 namenodes and 5 datanodes, each with 2 CPUs (Intel E5 2630) and 32 GB RAM. The whole cluster uses a switched 10 Gigabit Ethernet network and all nodes are running Ubuntu 14.04 with kernel 3.14. We set each data block has 3 replicas in HDFS. All experiments have been repeated 3 times and we present the arithmetic mean numbers.

5.2. Replication Cost

First of all, we measure the replication cost of NCluster, The cost of replication time is a metric to evaluate the performance penalty of high availability solution, if the penalty of metadata replication process is too high, the overall performance of HDFS cluster will be reduced dramatically. We take four typical update requests to measure the replication cost:

- create : to create a file
- rename : to rename an existing file
- mkdir: to create a directory
- delete: to delete a file

Note that the replication process contains two main parts: serialization (deserialization) of each parameters of operations a, and the transmission delay over network.

Copyright ⓒ 2016 SERSC

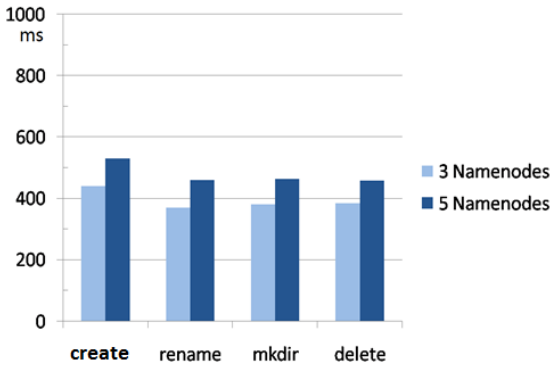

Figure 7. Replication Cost

We evaluate the replication cost over NCluster consists of 3 and 5 namenodes. Both of the NCluster have been tuned into sync mode. Figure 7 indicates the experimental result. The replication cost of create, rename, mkdir, delete operations are 435ms, 370ms, 381ms, 384ms in NCluster contains 3 namenodes respectively, the reason why the create operation takes longer time is that it contains 9 parameters, meanwhile the rest only have 2 or 3. The time spent in serialization and deserialization is relatively small(less than 10 millisecond) so we do not display them in the Figure7.

5.3. NCluster vs Zookeeper

As described in section 2, one could also make use of Zookeeper to build a cluster with multiple active namenodes. But the process of metadata replication would result in extensive disk I/O in every namenode's local disk, which is awfully inefficient.

We choose four typical write operations mentioned above and evaluate their metadata replication cost in Zookeeper schema, then compare it with NCluster contains 5 namenodes. The Zookeeper instances have been deployed on 5 namenodes to handle the synchronization of metadata files.

From Figure 8 we can see that the replication time of each operation spent by Zookeeper are 910ms, 796ms, 783ms , 779ms ,while NCluster only spends 530ms, 460ms ,463 ms, 458ms ,the Zookeeper schema has much higher replication cost , it spends more than 70% of time in compare with NCluster.

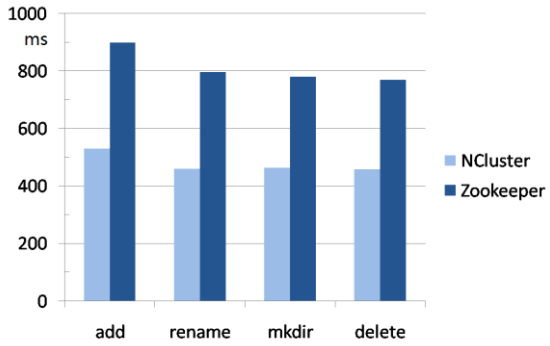

Figure 8. NCluster vs Zookeeper

5.4 Throughput

Throughput is another important metric when we evaluate the performance of a distributed system. We test the throughput of NCluster by measuring how many files HDFS clients can create within one minute when it is integrated with NCluster contains 5 namenodes.

 Copyright ⓒ 2016 SERSC

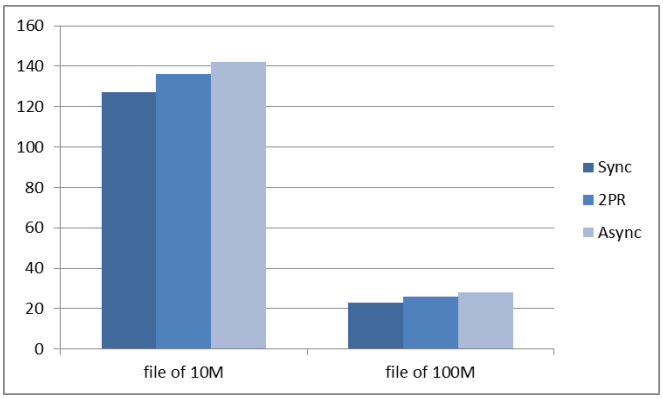

Figure 9. Throughput

As it can be seen in Figure 9, for files of 10M size, HDFS could create 127 times in one minute if NCluster is under Sync mode, whereas 136 and 143 times when it is under 2PR and Async Algorithm, for files of 100M size, NCluster creates 23, 26, 28 times respectively. Clearly the throughput under Async mode is higher than sync mode but the Async only guarantee the eventual consistency, HDFS clients may have a little chance to read inconsistent status of data during in certain period of time.

5.5 Scalability

Since NCluster utilizes multiple active namenodes for HDFS, it is essential to validate the scalability of NCluster .We test the read throughput of HDFS integrated with NCluster contains 3 to 9 namenodes to show the system scalability. We measure how many files HDFS clients can read within one minute.

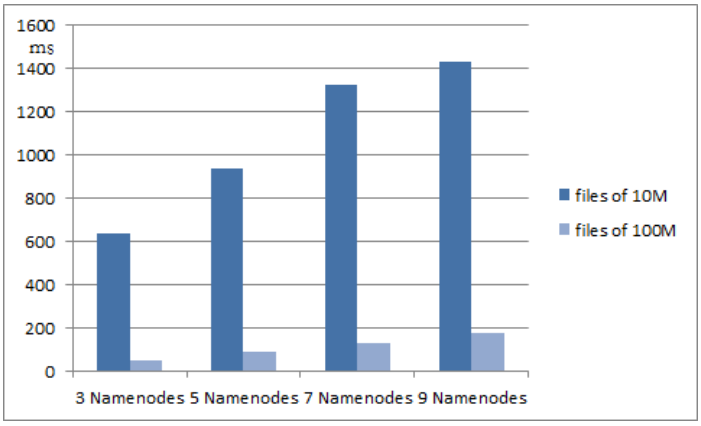

Figure 10. Scalability

As shown in Figure 10, as the number of namenodes goes up, the read throughput of HDFS clients increases almost linearly. Take file of 10M as an example, clients can read 641 times from HDFS with 3 namenodes in one minute, but if namenode scales up to 5 or 7, the read throughput will grow to 935 and 1323 times respectively.

5.6 Geo-Replication

We deploy the namenodes of NCluster at Tsinghua University in Beijing and SISDC (Suzhou International science-park Data Center) in Jiangsu to evaluate the replication time under high network delay circumstances. The average round-trip latencies between these 2 sites are approximate 200ms.

Copyright ⓒ 2016 SERSC

Table 2. % of Requests Finished when NCluster is Geo-Distributed

Time	N(1,2) with Sync	N(2,3) with Sync
	N(1,2) with 2PR	N(2,3) with 2PR
<250ms	5.12%	3.43%
	11.87%	7.37%
<500ms	29.56%	23.81%
	61.36%	52.67%
<750ms	56.11%	43.64%
	81.65%	73.76%
<1000ms	76.41%	66.09%
	100%	89.29%

In table 2, N (x,y) means we deploy x namenodes and y namenodes in Tsinghua and SISDC respectively, the primary namenode always resides in Tsinghua. We only compare the 2PR and Sync. We issue 20000 metadata update requests to NCluster to understand the benefits of 2PR under high network latency. Note that we only measure the time of metadata replication across Beijing and Jiangsu.

From the table we can see that the 2PR algorithms perform significantly better than the Sync, The cross domain network causes large latencies in the execution of the Sync. With the use of 2PR, NCluster are better suited for satisfying such network conditions. Take N(1,2) as an example, 2PR finished 100% of the metadata replication in one second whereas Sync only accomplished 76.41%, and evaluation on N(2,3) discovers the same fact.

6. Conclusions and Future Work

Hadoop HDFS is an open source project from Apache Software Foundation for scalable, distributed computing and data storage. Unfortunately, Hadoop has quite a little high availability support on HDFS namenode yet, existing solutions are far from satisfaction.

In this paper, we present a solution to improve the availability for namenode of HDFS .We build NCluster contains a small number of namenodes (typically 3 to 5), one namenode called primary, it handles all the write requests and a portion of read requests from HDFS clients, the rest of namenodes called hot standbys ,they are all read only. During the uptime, the primary replicates all the metadata updates to the hot standby namenodes continuously, we implement a pub/sub system to handle these replication processes quite efficiently, we also propose a novel replication algorithm to deal with the network delay when the namonodes are deployed in different areas. Experiments verify our solution's feasibility and effectiveness.

Right now, we use the strategy of random to load balance the multiple namenodes in NCluster. Our future work includes finding more effective and adaptive strategy to balance the read requests amongst multiple these namenodes based on workload and computing resource usage of the namenodes.

References

[1] Apache Hadoop. http://hadoop.apache.org/
[2] J. Dean and S.Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 04: 6th Symposium on Operating Systems Design and Implementation
[3] Apache HBase. http://hbase.apache.org/
[4] P. Hunt ,M. Konar, F. P. Junqueira and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. ATC 10: USENIXAnnual Technical Conference 2010
[5] K. Shvachko, H. Kuang, S. Radia, R. Chansler. The Hadoop Distributed File System. MSST10: 26th IEEE Symposium on Massive Storage Systems and Technologies.

 Copyright ⓒ 2016 SERSC

[6] E. Baldeschwieler and D. Cutting.. State of Hadoop. Hadoop Summit 2009.
[7] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. SOSP 03:proceedings of the 19th ACM Symposium on Operating Systems Principles
[8] M. Burrows. The Chubby Lock Service for Loosely Coupled Distributed Systems. OSDI 06: In Proceedings of the 7th Symposium on Operating Systems Design and Implementation
[9] L. Lampor. Paxos Made Simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) 51-58.
[10] M. Bar. The Linux VFS, Chapter 4 of Linux File Systems .McGraw-Hill 2001
[11] T. White. Hadoop: The Definitive Guide. O'Reilly Media2009.
[12] Facebook has the world's largest Hadoop cluster!
[13] http://hadoopblog.blogspot.com/2010/05/facebook-hasworlds-largest-hadoop.html
[14] D. Borthakur *e.g.* Apache Hadoop Goes Realtime at Facebook. SIGMOD 11: Proceedings of the 2011 International Conference on Management of Data.
[15] High Availability for Hadoop Distributed File System-2.0.5
[16] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithQJM.html
[17] Apache ActiveMQ. http://activemq.apache.org/
[18] M. Robert . Design Principles and Design Patterns.
[19] http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
[20] P.Maurice *et al.* Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems. 1990
[21] W.Vogels. .Eventually consistent.
[22] http://queue.acm.org/detail.cfm?id=1466448
[23] B. F. Cooper *et al*. Pnuts: Yahoo!'s hosted data serving platform. VLDB 2008: 34th International Conference on Very Large Data Bases
[24] B. Calder *et al*, Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. SOSP 11: 23rd ACM Symposium on Operating Systems:
[25] James Corbett *et al*. Spanner: Google's Globally-Distributed Database. OSDI 12: 10th USENIX Symposium on Operating Systems Design and Implementation

Authors

Zhanye Wang is currently a PhD candidate in Department of Computer Science and Technology of Tsinghua University. He received his MS degree from Beihang University in 2008. His research interests include massive storage, large-scale distributed system, and relational databases optimizations.

Tao Xu is currently a PhD candidate in Department of Computer Science and Technology of Tsinghua University. He received his MS degree from Tsinghua University in 2005. His research interests include massive storage, large-scale distributed system, and stream computing.

Dongsheng Wang received his PhD degree from Harbin Institute of Technology in 1995. He is now a professor in Department of Computer Science and Technology at Tsinghua University. He is a senior member of China Computer Federation, member of IEEE. His research interests include computer architecture, high performance computing, storage and file systems, and network security.

Copyright ⓒ 2016 SERSC

 Copyright ⓒ 2016 SERSC