Hadoop High Availability through Metadata Replication

Feng Wang
IBM China Research Laboratory
Beijing, 100193, China
wangfwf@cn.ibm.com

Jie Qiu
IBM China Research Laboratory
Beijing, 100193, China
qiujie@cn.ibm.com

Jie Yang
IBM China Research Laboratory
Beijing, 100193, China
yangyjie@cn.ibm.com

Bo Dong
Xi'an Jiaotong University
No.28, Xianning West Road
Xi'an 710049, China
dong.bo@mail.xjtu.edu.cn

Xinhui Li
IBM China Research Laboratory
Beijing, 100193, China
lixinhui@cn.ibm.com

Ying Li
IBM China Research Laboratory
Beijing, 100193, China
lying@cn.ibm.com

ABSTRACT
Hadoop is widely adopted to support data intensive distributed applications. Many of them are mission critical and require inherent high availability of Hadoop. Unfortunately, Hadoop has no high availability support yet, and it is not trivial to enhance Hadoop. Based on thorough investigation of Hadoop, this paper proposes a metadata replication based solution to enable Hadoop high availability by removing single point of failure in Hadoop. The solution involves three major phases: in initialization phase, each standby/slave node is registered to active/primary node and its initial metadata (such as version file and file system image) are caught up with those of active/primary node; in replication phase, the runtime metadata (such as outstanding operations and lease states) for failover in future are replicated; in failover phase, standby/new elected primary node takes over all communications. The solution presents several unique features for Hadoop, such as runtime configurable synchronization mode. The experiments demonstrate the feasibility and efficiency of our solution.

Categories and Subject Descriptors
C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS – *Design studies, Reliability, availability, and serviceability.* E.5 [**Data**]: FILES – *Backup/recovery.*

General Terms
Management, Design, Reliability.

Keywords
Hadoop, namenode, jobtracker, metadata, replication, high availability.

1. INTRODUCTION
The businesses today are facing tremendous challenges due to the complex applications and dramatic growth in data volumes. Hadoop, an open-source project developed for reliable, scalable, distributed computing and storage, has been widely adopted to support data intensive distributed applications [1]. More and more researchers, developers and users are interested in Hadoop with the purpose of building big data ecosystem such as Cloud based on it [2].

As a platform of computing and storage, availability of Hadoop is the foundation of applications' availability on it. It is necessary to keep full-time availability of platform for product environment, or else the inestimable loss will be caused. For example, on July 21, 2008, Amazon S3 stopped working for nearly eight hours, thus thousands of online stores using S3 service were down for hours too. This accident led to a great loss of revenues, and damaged reputations to S3 users [3].

Hadoop has tried some methods to enhance the availability of applications running on it, e.g. maintaining multiple replicas of application data and redeploying application tasks based on failures, but it doesn't provide high availability for itself. In the architecture of Hadoop, there exists SPOF (Single Point of Failure), which means the whole system gives up and becomes out of work caused by the failure of critical node where only a single copy is kept. SPOF of Hadoop thus is a huge threat to the availability of Hadoop.

To provide high availability for Hadoop, there are several challenges as follows.

(1) SPOF identification. Namenode and jobtracker are SPOF in Hadoop, and how to identify the critical component and state information more exactly to remove these SPOF is not an easy job.

(2) Low overhead. Achieve high availability needs additional time cost for runtime synchronization among different nodes, so a performance optimized solution for implementing high availability is necessary.

(3) Flexible configuration. To implement high availability for Hadoop, many configurable options should be considered to meet performance requirements of different workloads in different execution environments (e.g. network bandwidth and latency).

CloudDB'09, November 2, 2009, Hong Kong, China.

This paper analyzes the SPOF existing in critical nodes of Hadoop and proposes a metadata replication based solution to enable Hadoop high availability. The solution involves three major phases: in initialization phase, each standby/slave node is registered to active/primary node and its initial metadata (such as version file, file system image) are caught up with those of active/primary node; In replication phase which is the core phase of our solution, the runtime metadata (such as outstanding operations, lease states) for failover are replicated; in failover phase, standby/new elected primary node takes over all communications. In our solution, several unique features for improving availability of Hadoop are presented, such as online reconfigurable synchronization mode and corresponding adaptive decision method.

The rest of this paper is organized as follows. Section 2 investigates SPOF in Hadoop. Section 3 introduces our solution to enable Hadoop high availability. Section 4 evaluates the proposed solution. Section 5 includes the related works and section 6 concludes the whole paper.

2. SPOF OF HADOOP

Hadoop provides a distributed file system (HDFS) and a framework to run MapReduce application. Both of HDFS and MapReduce form the kernel of Hadoop [1, 4].

HDFS stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. A HDFS installation consists of a single namenode as the master node and a number of datanodes as the slave nodes. The namenode manages the file system namespace and regulates access to files by clients. The datanodes are distributed, one datanode per machine in the cluster, which manage data blocks attached to the machines where they run. The namenode executes the operations on file system namespace and maps data blocks to datanodes. The datanodes are responsible for serving read and write requests from clients and perform block operations upon instructions from namenode.

MapReduce is a computational paradigm, which divides the execution of application into many small fragments of work. Each of the fragments may be executed or re-executed on any machine in the cluster. As similar as HDFS, MapReduce also follows the architecture of master-slave. In Hadoop, the implementation of MapReduce has a single master node called jobtracker and several slaves nodes called tasktrackers. The jobtracker receives map/reduce jobs from clients, then puts them in a queue of pending jobs and schedules them on a first come first served basis. The jobtracker manages the assignment of map and reduce jobs' component tasks to tasktrackers. The tasktrackers execute tasks upon instructions from the jobtracker and also handle data motion between the map phase and the reduce phase.

According to the design of Hadoop architecture, it is clear that the namenode of HDFS and the jobtracker of MapReduce are both critical nodes which take very important roles in management of Hadoop. But because there is only a single copy of them in Hadoop, no matter which node is down, HDFS or MapReduce will be out of service immediately.

The target of our high availability solution is to remove these SPOF in Hadoop.

3. PROPOSED SOLUTION

In this paper, we propose a metadata (described in section 3.1) replication based solution to enable Hadoop high availability. The solution consists of three major phases: the first is initialization phase which initializes the execution environment of high availability; the second is replication phase which replicates metadata from critical node to corresponding backup node at runtime; and the third is failover phase which resumes the running of Hadoop despite the critical node is out of work.

The execution environment of high availability consists of the critical node and one or more nodes used for its backup. Our solution supports two types of topology architecture of nodes in execution environment: one is active-standby topology which consists of one active critical node and one standby node; the other is primary-slaves topology which consists of one primary critical node and several slave nodes. Within primary-slaves topology architecture, all nodes are reachable from others, so that slave nodes can communicate with each other for leader election described in section 3.4.1. The main difference between standby node and slave node is that the slave node takes a portion of read requests but the standby node does not, and it has little impact on our high availability solution

Both types of node topology architecture are suitable for medium amounts of files but higher volumes of read requests which are typical application scenarios of Hadoop. From the standpoint of architecture, active-standby topology is much simpler than primary-slaves topology, so unless otherwise noted, we take primary-slaves topology architectures as the representative in order to describe our solution more clearly.

To reduce performance penalty for replication, our solution only replicates metadata which are the most valuable management information for failover instead of a complete data copy stored in active/primary critical node. Note that all management information contained in jobtracker is stored in HDFS persistently and the information can be recovered for failover of jobtracker, so it is unnecessary to design specific metadata replication mechanism for jobtracker. Therefore, unless otherwise noted in this paper, we take the namenode as the representative to present our solution.

In the following sections, we present the description of metadata at first, and then present detailed information of each phase of our solution.

3.1 Metadata

Metadata are the most important management information replicated for namenode failover. In our solution, the metadata include initial metadata which are replicated in initialization phase and two types of runtime metadata which are replicated in replication phase.

The initial metadata include two types of files: version file which contains the version information of running HDFS and file system image (fsimage) file which is a persistent checkpoint of the file system. Both files are replicated only once in initialization phase, because their replication are time-intensive processes. Slave node updates fsimage file based on runtime metadata to make the file catch up with that of primary node.

The first type of runtime metadata is edit log which records the write operations submitted by file system clients. There are eleven types of write operations in HDFS, including: add, rename, delete, close, make directory, set replication, set permissions, set ownership, set generation stamp, update access time and set disk quota. Each operation is a piece of edit log. The metadata are replicated for tracking the write operations on HDFS fsimage. To avoid unbound growth of edit log file content, when the file size exceeds a predefined threshold, the standby node will load fsimage and edit log into memory and applies each write operations recorded by the edit log to update the fsimage file. The process is similar to the internal merge process of current secondary namenode [4].

The second type of runtime metadata is lease state. In HDFS, the file can only be written by a single writer in any time and namenode provides a lease to authorize a writer during a certain time interval [1]. For failover, lease state is valuable for guaranteeing operation consistency and should be recovered correctly. Lease state stays in memory and consists of the writer, file path and update time. A lease monitor takes charge to check if the last update time has expired, and the writer sends renew lease request periodically. We add a directory in slave node to store lease state information and a probe to intercept the renew lease request in primary node. When a renew request is received, the corresponding change of lease state is recorded by primary node. The metadata of lease state is helpful to do more accurate failover.

HDFS does not persistently store block mapping information, which contains the connection between a block and the datanode where it is stored. The block mapping information is constructed in memory based on the block lists sent by datanodes when they join Hadoop cluster and then be updated periodically based on block reports from datanodes. In our solution, we also don't take the block mapping information as the metadata, and the reconstruction of block mapping for failover is presented in section 3.4.2.

3.2 Initialization

The main tasks of initialization phase include node registration to register slave nodes and initial metadata synchronization to make initial metadata consistent between primary node and slave nodes.

3.2.1 Node registration

Node registration process makes the primary node know the initial states of existing slave nodes. In our solution, the registration process involves three major steps showed in Figure 1.

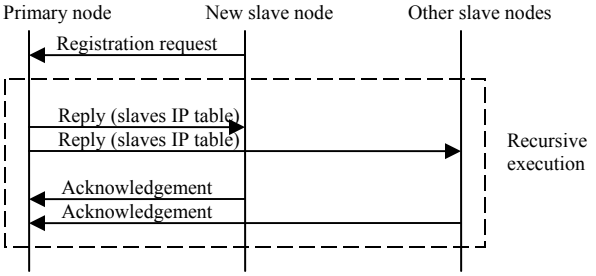

Figure 1. Slave node registration.

In Figure 1, the registration process begins with a slave node sending a registration request that contains the IP address information of itself to primary node. When primary node receives the request, it registers the slave node by writing the IP address information contained in the request into a slave IP address table and sends a reply message with the table contents to all slave nodes registered in the IP address table of primary node. Next, slave nodes receive the reply message from primary node, and then they update their local slave IP address table and send acknowledgements to primary node. Meanwhile, the primary node waits for acknowledgements sent by slave nodes which have been registered and checks if all acknowledgements from registered slave nodes have been received successfully. If it is done without timeout, the registration process is completed; otherwise, the primary node has to unregister the node whose acknowledgement is not received by removing corresponding IP address information from the slave IP address table, and then resends the content of updated table to all registered slave nodes. The process executes recursively till the whole registration process completes.

This registration process guarantees all of live slave nodes can be registered by the primary node during initialization phase. For active-standby topology architecture, the registration process is very simple, because there have no other nodes need to be registered except the standby node.

3.2.2 Initial metadata synchronization

When the registration process is completed, the initial metadata of slave nodes must be synchronized to catch up with those of primary node.

In our solution, the process of initial metadata synchronization involves three steps in sequence: version file checking, fsimage file checking and metadata synchronizing.

Firstly, in the version file checking step, primary node asks slave nodes to deliver their version file information and checks if the information is consistent to those of primary node, If no, e.g. no any name space ID (a unique identifier for the file system) exists in the information of slave node or its value is different from the name space ID kept in primary node, the slave node will be recorded as an inconsistent node by primary node.

Next, in the fsimage file checking step, primary node checks if there has fsimage file information of the slave node is inconsistent to primary, e.g. the latest update information of fsimage file. If there are any differences between fsimage file information of primary node and slave node, the slave node is considered as inconsistency, and then primary node will record the slave node as an inconsistent node.

Finally, in the metadata synchronizing step, primary node asks the inconsistent slave nodes for their version file or fsimage file to refresh the initial metadata stored previously. Primary node sends its initial metadata to inconsistent slave nodes and each slave node takes the received metadata as the consistent initial metadata.

This process keeps the consistency between the initial metadata of primary node and slave nodes, which provides correct foundation for the subsequent operations.

3.3 Replication

Replication is the core of our high availability solution, and it involves several unique features for Hadoop. The most important

is a flexible data synchronization mode and corresponding adaptive online decision method to meet performance requirements of different workloads in different execution environments.

In this section, taking the replication between primary node and one of slave nodes as an example, we introduce the architecture of replication at first, and then present more detailed descriptions of configurable synchronization mode.

3.3.1 Architecture

The main architecture of replication is presented in Figure 2.

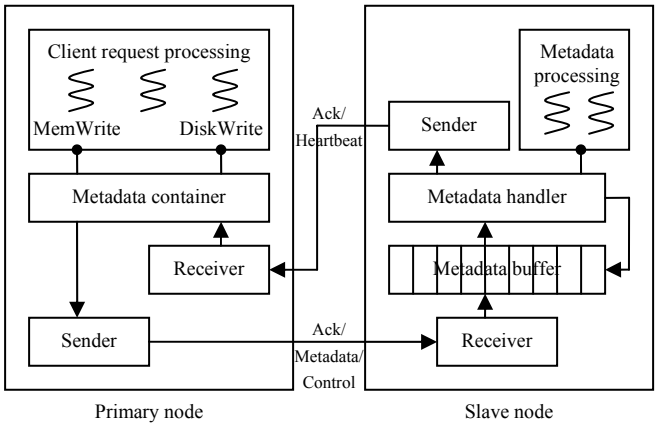

Figure 2. Architecture of replication.

In Figure 2, primary node and slave node have the same configurations of hardware and software. They communicate with each other by network. The connection-oriented TCP protocol is used in communication to ensure reliable data transfer.

In primary node, the metadata container component collects metadata from client request processing threads. It controls the in-memory processing (MemWrite), in-disk processing (DiskWrite) and commitment of multiple client requests simultaneously. Moreover, the metadata container involves an adaptive adjustor component (not showed in figure 2) which is one of the most important components of our Hadoop high availability solution. The adjustor can change synchronization mode (described in section 3.3.2) of replication based on runtime adaptive analysis to meet performance requirements of different workloads in different execution environments (such as network bandwidth and latency).

In slave node, the receiver component puts the received metadata to a metadata buffer. The metadata handler component gets metadata from metadata buffer and then handles the processing of received metadata including MemWrite, DiskWrite. There is an adjustor component in slave node too, and it changes synchronization mode of replication according to the control message sent from primary node. An additional function of this adjustor is to adjust the size of metadata buffer based on metadata processing performance and network communication speed.

There are two types of message are transferred from slave node to primary node: one is the acknowledgement message of received metadata; the other is the heartbeat message to indicate that the salve node is still alive. Meanwhile, there are three types of message are transferred in opposite direction: the first is the acknowledgement message of received heartbeat; the second is the metadata to be replicated; and the third is the control message to inform slave node change synchronization mode. These messages are sent/received by sender/receiver component in primary node and slave node.

3.3.2 Configurable synchronization mode

Synchronization mode is very critical to improve the performance of replication process. One of the most important features of our solution is to provide a configurable synchronization mode (showed in figure 3) to fit for diverse ranges of workload performance requirements and execution environments.

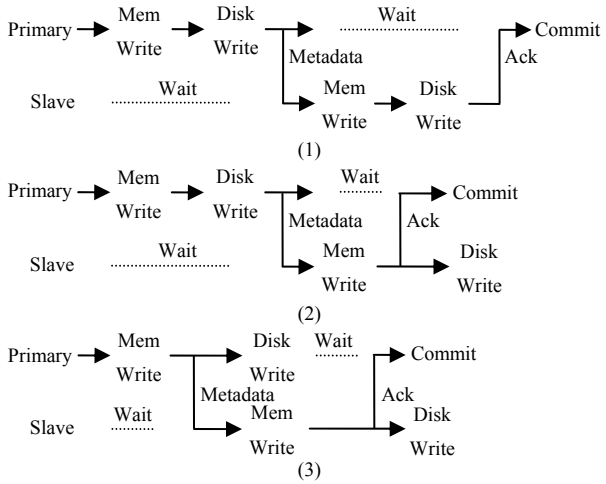

Figure 3. Synchronization mode.

In Figure 3, Mode (1) commits the client request on primary node only after relevant metadata have been written to disk on both primary and slave; Mode (2) commits the client request on primary node after relevant metadata have been written to disk on primary and received into memory on slave; and Mode (3) sends metadata before local disk write on primary node and commits the client request on primary node after relevant metadata have been written to disk on primary and received into memory on slave. Mode (1) and (2) guarantee the metadata are stored in primary node before they are sent to slave node, and Mode (3) has advantage on metadata transfer performance. To keep the consistency among nodes, we use a three-phase commit protocol with non-blocking capability [5].

We choose replication synchronization mode according to workload performance and network transfer speed. For a wide area network with low bandwidth, Mode (3) is recommended because transmission latency overlaps the time cost on metadata disk writing of primary. For a local area network with high bandwidth, the choice of synchronization mode is most often between Mode (1) and (2). Furthermore, the choice of Mode (1) or Mode (2) depends on the tradeoff between workload performance and synchronization protection. For example, Mode (1) has higher performance penalty for waiting for disk writing of metadata on slave node and Mode (2) has potential threat caused by the failure of slave node stores the metadata into disk.

In practice, waiting for acknowledgements sent from all slave nodes may result in large time overhead. So, to improve

performance, the primary node can commit request after it receives acknowledgements from a majority of slave nodes. This improvement needs some enhancements on synchronization modes in figure 3, e.g. assign sequence number for each metadata transfer and corresponding acknowledgement to handle the overdue acknowledgement which is received by primary node after the corresponding request has been committed.

In our solution, an adaptive method is provided to choose the most suitable synchronization mode for replication phase instead of a fixed configuration. Within adaptive method, network throughput threshold and workload performance threshold are configured by users who set the thresholds based on network environments and performance requirements. Analysis engine collects network throughput information and workload performance information at runtime. Meanwhile, the engine automatically makes the choice among three types of synchronization modes according to an adaptive analysis, i.e. collects data with a fixed time period and compares the average value of data with the threshold, then decides which mode is the best.

3.4 Failover

Once slave node has not received the acknowledgement of its heartbeat message for a long time which exceeds a predefined time interval threshold, the slave node considers that the primary node is out of work. Then, a failover process is started and a slave will act as the primary node at the end. Two steps for failover are leader election and IP address transition. If the active-standby topology is used, the former step is not necessary. Note that the failover process of namenode is similar to the process of jobtracker except that namenode needs an additional lease management step and some other delicate difference, such as block mapping information reconstruction of HDFS and job history recovery of MapReduce, so we give more detailed description in this section to the representative failover process of namenode.

3.4.1 Leader election

Leader election is a process of designating a slave node to take the place of primary node, and this process is a negotiation and it is handled by slave nodes automatically.

Within primary-slaves topology architecture, there may be a set of slave nodes that join into the competence of new primary node when the original primary node is out of work. Our solution introduces a node ordering mechanism to help slave nodes resolve this conflict.

In our solution, we order the slave nodes who are new primary node candidates by assigning them an increasing sequence number and make each slave node keep tracking the most recent sequent number it has seen so far. When a slave node believes the primary node is out of work and wants to become the primary node, it generates a unique sequence number at first. A simple method to generate sequence number is as follows: considering that there are n slave nodes (this information can be got by checking local salve IP address table), assigns each slave node r a unique id i_r between 0 and n-1, and the slave node r picks the smallest sequence number s larger than any it has seen such that $s \bmod n = i_r$. This method guarantees the number unique effectively. When sequence number is generated, slave node broadcasts the sequence number to all slave nodes in the slave IP address table except itself. When a slave node receives the message, it checks whether the primary node is out of work or not. If it finds that the primary node is still working, the salve node will respond a disagreement message to the sender immediately; otherwise, the slave node will continue to check if there has seen a higher sequence number. If the sequence number in the message is the highest, the salve node will reply with an agreement message, or else it replies with a disagreement message.

Next, if the slave node receives a majority of agreement messages replied from other slave nodes, the slave is qualified for being the primary node, then it sends a confirm message to tell slave nodes that it will take the role of primary node and the leader election is completed. Otherwise, the slave node has not received enough number of agreement messages, and a new round of leader election will be launched till a slave node satisfies the conditions to complete this election.

When leader election completes, the new primary node loads metadata stored in replication phase into memory for reconstructing the latest execution state of the old primary. For example, fsimage file is loaded and then each operation recorded in edit log is applied. Once the primary has reconstructed a consistent in-memory image, it creates new files for recording the following metadata. Only at this point, the primary is ready for listening requests from other nodes.

3.4.2 IP address transition

The namenode of HDFS is accessed through IP address. When the leader is elected, the new primary node changes its IP address to the IP address of the old primary node, so that it can takes over all communications with other nodes, e.g. datanodes, slave nodes. Meanwhile, other nodes will not find any change and they can access the primary node as directly as ever.

In our solution, primary node and slave nodes are placed on different servers which have own IP address. So, we take the IP address of primary node as a parameter and invoke Linux shell commands such as ifconfig to modify IP address information of the new elected primary node. Furthermore, it is necessary to change IP address information in the network configuration file directly, because the configuration set by ifconfig command will be lost when the machine is restarted. For jobtracker, the new primary node needs another modification in its hostname, it can be done by using Linux shell command such as hostname and modifying corresponding configuration files.

To speed up the failover process, we use a technique known as gratuitous ARP which is an ARP reply when there was no ARP request. The new primary node issues a gratuitous ARP reply message in order to trigger other nodes on the network to update their ARP table and to inform switches of the MAC address of the current primary node.

Within primary-slaves topology architecture, when the IP address transition is completed, the new primary node needs to initialize the remaining slave nodes again to guarantee the initial metadata consistency. According to the entries saved in slave IP address table, the new primary node sends a re-register message to the slave nodes except itself, and each slave node responds the

message by a registration request. Then, the subsequence process is the same as the initialization described in section 3.2.

For namenode of HDFS, an additional step is reconstructing block mapping information in memory based on block lists sent by datanodes. For jobtracker of MapReduce, an additional step is recovering job management information according to job history file stored in HDFS.

3.4.3 Lease management

In HDFS, to create or modify a file, client will first contact the namenode which will grant it a lease for writing the file. Client renews the lease periodically, and namenode checks the states of leases throughout the write process of client to look for if there is any lease expiration. Lease mechanism prevents a dead client from the long term resource holding.

In our solution, lease management is very important to namenode failover. If a client does not exhaust its lease before the primary is out of work, remaining time of the lease must be recovered when the primary failover is completed. This is necessary to keep system operation consistent. We provide a lease management by an approach similar as [6], but we need more information to support more accurate management. The latest update state of lease is recorded by primary namenode and replicated to slave nodes as described in section 3.1. When the slave node finds the primary is out of work, it records the time as the primary down time. When the failover process completes and client contacts with the new primary, the new primary will calculate the difference between the old primary down time and the latest update time of client lease as the elapsed lease time of client and check if the elapsed lease time is smaller than a predefined limit or not. If yes, the new primary uses the difference between the elapsed lease time and the predefined limit as the remaining lease time of corresponding client, and the client still has lease to write the file in this time interval.

4. EXPERIMENTS

Two experiments are used to evaluate the feasibility and efficiency of our Hadoop high availability solution: one is to measure the failover time for critical node; the other is to measure the time overhead induced by runtime replication. Note that we take the failover and replication of namenode as the representative in the experiments; the corresponding processes of jobtracker are very similar to namenode.

4.1 Experiment Environment

The HDFS cluster in our experiment consists of 5 PC machines. The active-standby topology is used; specifically, there are one active namenode, one standby node and three datanodes.

The active namenode and the standby node are installed in the same hardware and software configuration, which is Intel Pentium 4 CPU 3.2GHz, 1.5G DDR 400MHz memory, 1T disk and SUSE Linux Enterprise Server 10.2 with the kernel of version 2.6.12. All datanodes have similar hardware and software configuration as namenode, except that they have only 1G DDR 400MHz memory. All machines are interconnected with 1.0Gbps Ethernet network.

Hadoop 0.20.0 is installed in all machines. Because the number of file blocks affects HDFS significantly, to evaluate our solution with different storage pressure, multiple file sets are auto-generated. These files sets contain 5000, 10000, 50000 and 100000 files respectively. The size of each file is smaller than 64M, which is the default size of a HDFS data block.

4.2 Failover Time

Failover time is a metric to evaluate the performance of failover process. The failover process begins at the time when active/primary namenode is out of work and ends at the time when standby/new elected primary node takes over the original active/primary node.

Generally, the time interval of failover consists of four portions: leader election time, IP address transition time, network transfer time and block mapping construction time. Among these portions, the time of leader election maybe have significantly huge fluctuation because there are unpredictable conflicts during the election process; the time of IP address transition can be neglected because it only costs few time on network configuration processing; the time of block mapping construction is related to the number of blocks which are reported in block reports sent from datanodes. There is no leader election process in our experiments environment, so we take the sum of block mapping construction time and network transfer time as the failover time.

To start the experiment, we upload the file set to HDFS cluster at first. Each file is stored in a data block and each block has three replicas which are distributed to three datanodes. Next, we shut down the active namenode by pulling its power cord, then the failover process begins to execute till the standby node replaces the role of original active namenode. We record the time of block mapping construction and network transfer, then compute the failover time by averaging the sum of the two values. We repeat twenty times of the experiment with each file set and take the average as the failover time of HDFS. The experiment results are illustrated in figure 4.

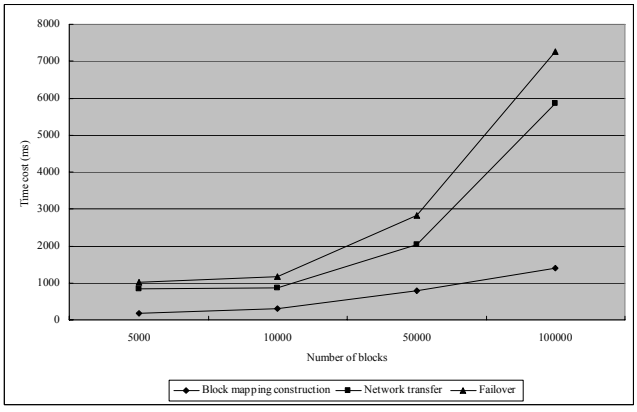

Figure 4. Failover time cost.

In figure 4, the time of failover varies from nearly 1 second to more than 7 seconds, and increases as the number of blocks increases. The other two lines in figure 4 have similar trends, because more blocks means more block mapping information to be transferred to namenode.

Theoretically, the time cost of failover is in direct proportion to the size of file sets, but it is not very clear in figure 4. The reason is batch processing of block mapping information reduces the additional penalty in processing big volume data.

Within failover time, the proportion of block mapping construction time is always much smaller than network transfer time, because the process of block mapping construction consists of in-memory operations and the structure of each block mapping information is simple. Therefore, the network transfer is bottleneck of failover process.

4.3 Replication Time Cost

Replication time cost is a metric to evaluate the performance penalty of high availability solution. In our solution, the replication of initial metadata (i.e. fsimage file and version file) is executed only once in initialization phase, so it has little impact on runtime performance of HDFS. We focus on the time overhead of runtime metadata replication which runs concurrently with the normal execution of namenode. Because the metadata processing of active/primary namenode must be synchronized with the standby/slave nodes at runtime, if the penalty of replication process is too high, the performance of HDFS will reduced dramatically. In our experiments, the synchronization mode (1) described in section 3.3.2 is always be chosen by adaptive analysis engine, because the network bandwidth is high enough. So, the process of metadata processing begins at the time when client request is processed in memory of active node and ends at the time when the active node receives the acknowledgement from standby node. In this metadata processing interval, the time consumed by memory writing and disk writing in active namenode is also required in normal execution process without replication.

In our experiments, we upload the file set to HDFS cluster at first, and then create a new file. Creating a file involves two write operations: add and close. Both write operations generate metadata to update namespace information stored in namenode. Additionally, the add operation requests lease from namenode and the close operation returns lease to namenode, therefore corresponding lease states are collected as metadata too. All metadata are replicated to standby node. We record time cost of major steps in replication process includes memory writing and disk writing in active namenode and standby node and network communication. For comparing metadata processing performance between replication process and normal process, we also record the time overhead for file creation in HDFS without replication process. The normal process has only two steps that are memory writing and disk writing in namenode. We repeat fifty times of the experiment with each file set and compute the average. The experiment results are illustrated in table 1.

Table 1. Metadata processing performance comparison between replication process and normal process

Number of blocks	Process type	Mem. write in namenode (ms)	Disk write in namenode (ms)	Mem. write in standby node (ms)	Disk write in standby node (ms)	Network communication (ms)	Metadata processing (ms)
5000	Replication	0.043	1.581	0.351	1.247	3.649	6.871
	Normal	0.084	2.042	-	-	-	2.126
10000	Replication	0.129	1.636	0.431	1.330	3.426	6.952
	Normal	0.105	2.577	-	-	-	2.682
50000	Replication	0.151	2.032	0.505	1.257	3.651	7.596
	Normal	0.127	2.265	-	-	-	2.392
100000	Replication	0.165	1.899	0.529	1.400	3.499	7.492
	Normal	0.177	2.185	-	-	-	2.362

In table 1, the metadata processing time overhead of replication process is nearly two times longer than that of normal process. The most time-intensive step is network communication which spends half of whole replication time, so the network communication is bottleneck. A suitable synchronization mode is helpful to resolve this problem, e.g. use synchronization mode (3) described in section 3.3.2.

The time cost of in-memory processing in standby is always much longer than that of active namenode, because it needs message parsing to get the metadata when the standby node receives packets from network and additional lease information processing is time-consuming too.

According to the results in table 1, the size of file sets stored in cluster is irrelevant to the replication time cost. Because only edit log entry and lease state record is transferred as the metadata at each time, the size of metadata transferred is small and nearly invariant.

5. RELATED WORKS

High availability is an emerging topic in Hadoop community [2]. To our best known, there is no solution to improve the availability of Hadoop effectively.

Hadoop provides a secondary namenode [1, 4], which unfortunately does not act as a hot backup daemon for namenode. Instead, it is mainly used for periodically merging the metadata contained in namenode to prevent the data size from becoming too large.

A subproject of Hadoop, named Zookeeper [7], supports replication among a set of servers and provides a coordination mechanism for leader election among the servers, but it focuses on

providing a coordination service for distributed applications instead of a high availability solution.

ContextWeb experiments a high availability solution of Cloudera Hadoop [8]. The solution primarily makes use of DRBD from LINBIT and Heartbeat from Linux-HA project, but it is not optimized for availability and performance of Hadoop, e.g. several unnecessary data have to be replicated.

Replication plays an important role in our Hadoop high availability solution. Although quite a few replication mechanisms are available in mission-critical applications especially the database applications, these mechanisms are not suitable for Hadoop high availability.

E. Sorensen adds hot standby replication functionality to the Apache Derby [9], but it does not support multi-threading to deliver replication messages. So, the solution cannot be used for parallel processing of large scale data in Hadoop.

Berkeley DB [10] has a well-defined replication mechanism, but it targets at the database management system only. Users have to spend a lot of time redesigning the replication framework if they would like to use Berkeley DB replication for applications other than database.

MySQL [11] presents several replication solutions used in many different environments for a range of purposes, but it cannot reconfigure replication process dynamically. Moreover, MySQL has no official solution for failover.

IBM DB2 HADR (High Availability Disaster Recover) [12] is capable of adjusting configuration of replication process for better performance at runtime. The runtime configuration depends on a simulator to estimate the performance of replication happened before. It is not an effective approach to tune replication process according to the actual execution scenario.

6. CONCLUSIONS

Both namenode and jobtracker are critical nodes in Hadoop. In this paper, in order to enable Hadoop high availability, we present a metadata replication based solution to remove the SPOF of namenode and jobtracker. In our solution, when an initialization phase consists of standby/slave nodes registration and initial metadata synchronization is completed, a replication phase is executed at runtime. To reduce performance penalty, we only replicate metadata which includes outstanding operations and lease states for failover in future. During the failover phase, standby/new elected primary node recovers all metadata and takes over all communications to resume the execution of Hadoop. Different from existing replication technologies of database, our solution caters to the specific requirements of Hadoop, e.g. provides adaptive method to reconfigure synchronization mode of replication at runtime.

Our experiments illustrate that our solution enable Hadoop high availability effectively. Within our experiment environment, it took less than 10 seconds for the whole failover process during which the new active node recovered metadata for three datanodes that each has 100000 data blocks.

Our future work includes researching more effective adaptive algorithms to adjust configuration of replication process (such as synchronization mode, metadata buffer size) and testing our solution in Hadoop cluster with larger number of datanodes. Additionally, we will introduce into Hadoop more high availability technologies other than the replication based one.

7. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/

[2] E. Baldeschwieler and D. Cutting. 2009. State of Hadoop. In Hadoop Summit 2009 (Santa Clara, US, June 10, 2009).

[3] A. Stern. 2008. Amazon S3 Down. July 20, 2008. http://www.centernetworks.com/amazon-s3-down-july-2008

[4] T. White. 2009. Hadoop: The Definitive Guide. O'Reilly Media, Inc. June 2009.

[5] D. Skeen and M. Stonebraker. 1983. A Formal Model of Crash Recovery in a Distributed System. IEEE Transactions on Software Engineering. Vol. 9, Issue 3 (May 1983), 219-228. DOI = http://doi.acm.org/10.1109/TSE.1983.236608

[6] M. Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (Seattle, WA, USA, November 06-08, 2006). OSDI '06. USENIX Association, Berkeley, CA, 335-350.

[7] Apache Zookeeper. http://hadoop.apache.org/zookeeper/

[8] C. Bisciglia. Hadoop HA Configuration. Jul. 22, 2009. http://www.cloudera.com/blog/2009/07/22/hadoop-ha-configuration/

[9] E. Sorensen. 2007. Derby: Replication and Availability. MS Thesis. Norwegian University of Science and Technology. June 2007.

[10] H. Yadava. The Berkeley DB Book. Apress. Oct. 2007.

[11] MySQL. http://www.mysql.com/

[12] Torodanhan. 2009. Best Practice: DB2 High Availability Disaster Recovery. Apr. 1, 2009. http://www.ibm.com/developerworks/wikis/display/data/Best+Practice+-+DB2+High+Availability+Disaster+Recovery