OUKA
The University of Osaka
Institutional Knowledge Archive

Title	A Study on Partial Snapshots and Coordinator Replication for Fault-Tolerance of Large-scale Distributed Systems
Author(s)	金, 鎔煥
Citation	大阪大学, 2015, 博士論文
Version Type	VoR
URL	https://doi.org/10.18910/52013
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

A Study on Partial Snapshots and Coordinator Replication for Fault-Tolerance of Large-scale Distributed Systems

Submitted to

Graduate School of Information Science and Technology

Osaka University

January 2015

Yonghwan KIM

List of Major Publications

Journal Papers

1. Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa, "A concurrent partial snapshot algorithm for large-scale and dynamic distributed systems," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 1, pp. 65–76, 2014.

2. Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa, "A distributed and cooperative NameNode cluster for a highly-available Hadoop distributed file system," *IEICE Transactions on Information and Systems*, vol. E98-D, no. 4, 2015. (to appear).

Conference Papers

3. Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa, "Brief Announcement: A concurrent partial snapshot algorithm for large-scale and dynamic distributed systems," in *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security (SSS 2011)*, LNCS 6976, pp. 445–446, 2011.

4. Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa, "A distributed NameNode cluster for a highly-available Hadoop distributed file system," in *Proceedings of the 33th International Symposium on Reliable Distributed Systems (SRDS 2014)*, pp. 333–334, 2014.

Technical Reports

5. Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa, "A Distributed and Cooperative NameNode Cluster for a Highly-Available Hadoop Distributed File System," in *IPSJ High Performance Computing Symposium (HPCS2013)*, pp. 93, 2013.

6. Asaha Ishii, Yonghwan Kim, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Evaluation of Hadoop system consisting of virtual machines on multi-core CPUs (in Japanese)," in *IPSJ SIG Technical Reports*, vol. 2012-HPC-136(20), pp. 1-7, 2012.

Abstract

A distributed system consists of autonomous computing entities (i.e., computers, usually called *nodes*) that are connected with each other by asynchronous communication links. Nodes communicate with each other by exchanging messages to solve a problem. A fault-tolerant system is one that can continue the correct execution even in the presence of hardware and/or software faults. Fault-tolerance is the attribute that enables a system to achieve fault-tolerant operation. Because distributed systems are prone to be fault, fault-tolerance is one of the most important issues in designing reliable distributed systems.

The fault-tolerance of the distributed systems can be guaranteed and implemented using redundancy. There are three kinds of redundancy: *information redundancy*, *time redundancy*, and *physical redundancy*. With information redundancy, some extra bits are added for recovery of data when some of data are corrupt or lost. Time redundancy makes a system to execute the same action again, if necessary. Physical redundancy is implemented by adding extra computing entities (i.e. nodes) and makes it possible for the system as a whole to tolerate the loss or malfunctioning of some components.

In this dissertation, we introduce two methodologies of the fault-tolerance with some practical applications. In Chapter 2, we discuss checkpoint-rollback recovery which is a universal method for restoring distributed systems after faults using time redundancy approach. To realize checkpoint-rollback recovery, recording a consistent global state is required for the checkpointing, and it can be taken by a specific algorithm, called a *snapshot algorithm*. Therefore, checkpointing requires a sophisticated snapshot algorithm especially if the systems are large-scale, since repeatedly taking a global snapshots of the whole system requires unacceptable communication cost. As a sophisticated snapshot algorithm, a partial snapshot algorithm has been introduced that takes a snapshot of a subsystem consisting only of the nodes that are communication-related to the initiator instead of a global snapshot of the whole system. In Chapter 2, we modify the previous partial snapshot algorithm to create a new one that can take a partial snapshot more

efficiently, especially when multiple nodes concurrently initiate the algorithm. And we show from some experiments that the proposed algorithm greatly reduces the amount of communication needed for taking a partial snapshot.

In Chapter 3, we introduce a new architecture of Hadoop distributed file system (HDFS) using the physical redundancy approach. Recently, HDFS has attracted much attention from engineers and researchers as an emerging and effective framework for Big Data, because it can manage a huge amount of data with high performance and reliability using only commodity hardware. However, HDFS consists of a single coordinator (known as a master node), called *NameNode*, and many worker nodes, called *DataNodes*, and NameNode manages the entire namespace of a file system. This causes the single point of failure (SPOF) 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. In Chapter 3, we propose a new architecture of HDFS which allows multiple NameNodes to resolve these problems. We also evaluate our proposed system's performance and the effect of the load balancing compared with related works.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

A distributed system consists of autonomous computing entities (i.e., computers, usually called *nodes*) that are connected with each other by asynchronous communication links[4]. Nodes communicate with each other by exchanging messages and they cooperate to accomplish the objectives of the system. There are several fundamental issues in designing distributed algorithms (or protocols) on the distributed systems, for example, scalability, availability, efficiency, resource sharing, and fault-tolerance.

A *fault*[7] is a physical defect, imperfection, or flaw that occurs within some hardware or software component. A fault is blemish, weakness, or shortcoming of a particular hardware or software component. An *error* is the manifestation of a fault. Specifically, an error is a deviation from accuracy or correctness. Finally, if the error results in the system performing one of its functions incorrectly then a system *failure* is also the performance of some function in a subnormal quantity or quality.

In this dissertation, we mainly focus on fault-tolerance of distributed systems. In distributed systems, faults of one or more nodes may affect the entire system, for example, a single fault at a node may cause system stop. Thus as the number of nodes is increased, distributed systems are more prone to be fault. Therefore, fault-tolerance is one of the most challenging problems in distributed systems, and an important issue which needs to be solved toW design a large-scale distributed system.

There are many levels of faults, e.g. hardware, software, or communications, and they are usually classified into the following four types [27, 16].

1. **Crash fault.** When a node undergoes a crash fault, it stops permanently.

2. **Omission fault.** In a network such that nodes send and receive messages with each other, a receiver node does not receive some of the messages sent to it (receive omission), or some of the messages does not be sent even if the sender node tried to send them (send omission) when it undergoes an omission fault.

3. **Transient fault.** A transient fault changes the state of a node arbitrarily by changing their memory contents arbitrarily.

4. **Byzantine fault.** A Byzantine fault makes the process behave arbitrarily. This model is the strongest model of all the fault models.

In this dissertation, we focus on the crash fault only. This implies that a node may be stopped at any time during the system execution, and it keeps stopping (no local computations and no communications).

There are many methodologies for fault-tolerance depending on the distributed system model (e.g. mobile ad-hoc networks, sensor networks, or vehicle networks) and the approaches (i.e. how to guarantee the fault-tolerance to the system).

To guarantee the fault-tolerance of the distributed systems, the best it can do is to try to hide the occurrence of faults from other nodes. This technique is called *fault masking*, and it can be implemented using redundancy. There are three kinds of redundancy [4]: *information redundancy*, *time redundancy*, and *physical redundancy*.

With information redundancy, some additional data bits are added to recovery from garbled bits. For example, a Hamming code adds some bits to transmitted data to recover from noise on the transmission line.

With time redundancy, an action is performed, and then, if needed, it is performed again. If a transaction aborts, it can be redone with no harm. The time redundancy technique is helpful when the faults are transient faults. A checkpoint-rollback recovery is a typical technique to implement the time redundancy. To perform again, the system stores its state periodically to the specific non-volatile memory (stable storage). In the distributed systems, a specific algorithm is required to store its state. We will discuss such a algorithm in Chapter 2 of this dissertation.

With physical redundancy, extra computing entities (i.e. nodes) are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. For example, extra nodes can be added to the system so that if a small number of nodes crash, the system can

still function correctly. In other words, by replicating processes, a high degree of fault-tolerance may be achieved.

The physical replication of hardware is perhaps the most common form of redundancy used in systems. It is used in biology (mammals have two eyes, two ears, two lungs, etc.), aircrafts (have four engines but can fly on three), and sports (multiple referees in case one misses an event). There are three basic forms of hardware redundancy: *passive*, *active*, and *hybrid* [7].

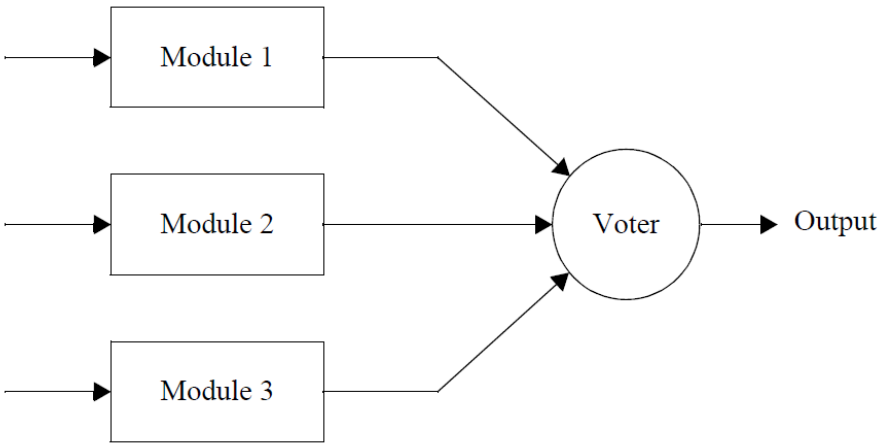


Figure 1.1: Basic concept of Triple Modular Redundancy (TMR)

Passive technique uses the concept of fault masking to hide the faults and it is designed to achieve fault-tolerance without requiring any action. The passive hardware redundancy relies upon voting mechanisms to mask the occurrence of faults. Most passive approaches are developed around the concept of majority voting. The most common form of passive hardware redundancy is called *triple modular redundancy* (TMR). The basic concept of TMR is to triplicate the hardware and perform a majority vote to determine the output of the system. If one of the modules becomes faulty, the two remaining fault-free modules mask the results of the faulty module when the majority vote is performed (Fig. 1.1).

Active technique achieves fault-tolerance by detecting the existence of faults and performing some actions to remove the faulty hardware from the system. In other words, the active techniques require that the system perform reconfiguration to tolerate faults. It attempts to achieve fault-tolerance by fault detection, fault location, and fault recovery.

Hybrid technique combines the attractive features of both the passive and the active approaches. Fault masking is used to prevent the system from producing errneous results, and fault

detection, fault location, and fault recovery are used to reconfigure the system in the event of a fault. Hybrid redundancy is usually very expensive in terms of the amount of hardware required to implement a system; consequently, hybrid redundancy is most often used in applications that require extremely high integrity of the computations.

In this dissertation, we introduce a new architecture of Hadoop distributed file system (HDFS) [26] using hardware redundancy technique in Chapter 3. HDFS (decribe more particularly in Chapter 3) is a distributed file system which is applicable to huge amount of data. HDFS has only one coordinator (known as a master node) which stores all metadata of the file system, thus it becomes a single point of failure (SPOF). We attempt resolving SPOF problem installing additional servers (NameNodes) by the (active) hardware redundancy approach. Moreover, with this hardware redundancy technique, we resolve the scalability problem and realize the load balancing.

As we mentioned in this section, we discuss two methodologies for fault-tolerance, a snapshot algorithm for checkpointing-rollback recovery (time redundancy technique) and server replications on HDFS (physical redundancy technique), with some practical applications.

1.1.1 A Snapshot Algorithm for Large-scale Systems

Checkpoint-rollback recovery is a universal (i.e., application-independent) method for fault-tolerance. It can make it possible to restore a distributed system state from faults. System periodically records its state into non-volatile storage (checkpointing) and it recovers its state using the stored state when faults occur (rollback).

From a system-wide viewpoint, each node has to recover its state so that the states of all nodes form a *consistent* global state [29] (or a global state without an orphan message, which is a received message but not be sent by the sender). To avoid the domino effect of rollbacks [6] (or an unbounded chain of rollbacks for attaining a consistent global state), nodes have to store their states cooperatively so that the stored checkpoints form a consistent global state of the whole system. The consistent global state formed by the recorded checkpoints of the nodes is called a *snapshot*, and the algorithm with which nodes record their checkpoints cooperatively is called a *snapshot algorithm*. There are many criteria for snapshot algorithms, a snapshot domain, frozen or non-frozen, concurrent executability, and applicability to dynamically changing topology.

Many sophisticated snapshot algorithms have been proposed. As the size of a distributed system increases, the efficiency of snapshot algorithms becomes more important. In large-scale distributed systems, repeatedly taking global snapshots of the whole system incurs unacceptable

communication cost. Another important requirement of snapshot algorithms is adaptability to dynamic distributed systems like web services [53], for example, RosettaNet [40], where nodes can freely join and leave the system at any time.

In Chapter 2, we introduce a new snapshot algorithm which is suitable for large-scale and dynamic distributed systems. We modify the previous partial snapshot algorithm to create a new one that can take a partial snapshot more efficiently, especially when multiple nodes concurrently initate the algorithm. And we show from some experiments that the proposed algorithm greatly reduces the amount of communication needed for taking a partial snapshot.

1.1.2 A Cooperative NameNode Cluster for HDFS

Hardware redundancy is intuitive methodology of fault-tolerance. It can be implemented using a replication of servers. Replication is a powerful technique for reliability, fault-tolerance, or accessibility [39].

To guarantee fault-tolerance, a node can create its replica which can have a role of the orignal node when it fails. Moreover, replication can also be used for a load balancing. Instead of single node processing, two or more replicated nodes can process a multiple requests concurrently.

In this dissertation, we apply this technique to Hadoop distributed file system (HDFS) [26]. HDFS is a distributed file system which can manage a huge amount of data with high performance and reliability using only commodity hardware. HDFS consists of only one coordinator (known as a 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 atore actual data blocks in their local storages. However, this architecture leads to some problems: single point of failure (SPOF), scalability problem, and load balancing. We attempt resolving these problems using redundancy (replication) technique. We also evaluate our proposed system's performance and the effect of the load balancing compared with related works.

1.2 Organization of This Dissertation

This dissertation consists of four chapters. In Chapter 2, we introduce our proposed snapshot algorithm for efficient checkpointing. In Chapter 3, we describe our proposed fault-tolerant architecture using replicated servers and their cooperations on Hadoop distributed file system. We conclude this dissertation in Chapter 4.

Chapter 2

A Snapshot algorithm for Large-scale Systems

2.1 Introduction

In this Chapter, we present a new snapshot algorithm which is suitable for large-scale and dynamic distributed systems. Our proposed snapshot algorithm can be utilized adaptively for *checkpoint-rollback recovery* [2, 37, 50], which is a universal method for restoring a distributed system from faults. Each nodeperiodically records its local state into non-volatile storage and it recovers its state from the recorded state when faults occur. The recorded state of a node is called a (local) *checkpoint*, and restoring the node state to the checkpoint is called a *rollback*.

2.1.1 Contribution

Moriya and Araragi [41, 42] proposed a distributed algorithm called the *Sub-SnapShot (SSS)* algorithm which is a partial snapshot of a subsystem consisting only of communication-related nodes. SSS algorithm can take a partial snapshot independently of the scale of the whole system, thus it can be applicable to large-scale distributed systems. However, SSS algorithm can guarantee its valid operation only when one node initiates SSS algorithm at the same time. This implies that SSS algorithm can not support the concurrent execution by multiple initiators.

We resolve this restriction of SSS algorithm by presenting a new partial snapshot algorithm called *Concurrent Sub-Snapshot (CSS)* [54] algorithm that can efficiently take a partial snapshot even when multiple nodes concurrently initiate the algorithm. CSS algorithm successfully realizes

an efficient solution for the concurrent execution problem by consistently merging two or more concurrent SSS algorithm executions when a collision occurs on a node. The initiators of the concurrent executions communicate with each other to cooperatively combine the executions, but the overhead for the combining process is small. The experimental results prove that CSS algorithm requires very small cost for resolving the restriction.

2.2 Related works

In a fault-tolerant distributed system, backward error recovery requires that the system correctly records its state onto stable storage. In particular, a (distributed) snapshot algorhtm is required for recording a consistent global state. When the system recovers using its checkpoint, a domino effect (one rollback causes other rollback) can be occurred if there is an orphan message, which is a received message but no node sent it, in the checkpoint. A domino effect may expend a huge amount of computational (or communication) cost, furthermore, many events, which are occurred before, can be lost. Thus, a consistent global state has to contain no orphan message.

Many sophisticated snapshot algorithms, which take a consistent global state, have been proposed. Chandy and Lamport [20] proposed a distributed algorithm for taking a (global) snapshot of a whole system. This snapshot algorithm assumes that all the channels guarantee the FIFO property. Lai and Yang [43] and Mattern [14] proposed snapshot algorithms for distributed systems with non-FIFO channels. All the above algorithms allow easy implementation and high efficiency, but still require $O(n^2)$ messages since every pair of neighboring nodes exchanges messages, where n denotes the number of nodes in the system. Thus, these algorithms are not applicable, in practice, to large-scale distributed systems. While further research reduces complexity by simplifying the taking of snapshots (e.g. $O(n \log n)$) [42, 34, 43, 35, 36], the scalability of snapshot algorithms remains critical. Moreover, it is impossible to apply these algorithms to dynamic distributed systems where nodes can join and leave the system at any time, which is common in a system like web-services.

Communication-induced checkpointing is an alternative approach to scalable snapshot algorithms [19, 33, 9, 28]. Not all nodes are requested to record their checkpoints in the algorithms, but some are, depending on the communication pattern. For distributed applications based mainly on the local coordination of nodes, communication-induced checkpoint algorithms can reduce the communication and time required for recording the checkpoints of nodes. However, the recorded checkpoints provide no guarantee that a consistent global state can be formed from the latest checkpoints of the nodes. This forces each node to keep multiple checkpoints in its non-volatile

storage and requires a sophisticated method to find a set of checkpoints of nodes that forms a consistent global state.

To achieve scalability of snapshot algorithms, Moriya and Araragi [41, 42] introduced a *partial snapshot*[1] of a subsystem consisting only of communication-related nodes and showed that the whole system can recover from faults by restoring each node of the subsystem to the latest record state. For taking a partial snapshot, Moriya and Araragi [41, 42] also proposed a distributed algorithm called the *Sub-SnapShot (SSS)* algorithm. In practical distributed applications, the number of nodes in a communication-related subsystem is expected to be much smaller than the total number of nodes in the whole system. Therefore, SSS algorithm can take a partial snapshot efficiently and thus makes the checkpoint-rollback recovery applicable to large-scale distributed systems. Another important advantage of SSS algorithm is that it is applicable to dynamic distributed systems (where nodes can join and leave the system), because each node needs no a priori knowledge of its adjacent nodes and can augment the knowledge during the execution of SSS algorithm. Koo and Toueg[37] introduced a communication-induced checkpoint algorithm applicable to dynamic distributed systems, but it has to suspend the executions of applications while taking checkpoints to guarantee consistency. On the other hand, SSS algorithm frees itself from this suspension by an elaborate operation on communication-relation. Lastly, SSS algorithm satisfies the strong consistency requirement just as Chandy and Lamport [20] does. That is, if a sender node records sending of a message in its stored local state, then the receiver node must record the message as a received or an in-transit one.

Although SSS algorithm has the above advantages, its drawback is an inability to guarantee the strong consistency of the partial snapshot it takes when multiple nodes concurrently initiate SSS algorithm and the subsystems communication-related to the initiators overlap. Each node may need to confirm its local state at any time independently from other nodes and this causes that two or more nodes initiate the snapshot algorithms (especially in the large-scale distributed system) at the same time. Thus, resolving this problem (two or more snapshot algorithms overlap) is necessary to apply a partial snapshot algorithm to large-scale distributed system. We call this problematic situation a *collision* of SSSs. The collision of concurrent initializations of SSS algorithm can be obviously resolved when concurrent initiations are handled sequentially one by one, but algorithm efficiency is severely degenerated. The algorithm proposed by Spezialetti[32] and its improved variant by Prakash[38] solved the problem of concurrent initiations. But their methods still target the creation of a global snapshot, and while it may be partitioned into pieces

[1] In [32], part of a global snapshot is also called a *partial snapshot*, but the notion is completely different from that in SSS algorithm, which is not a part of a single static global snapshot.

for efficiency, they are not applicable to dynamic situations.

2.3 Preliminaries

2.3.1 System Model

We consider distributed systems consisting of nodes (or processes). Nodes share no common memory or storage and communicate with each other asynchronously by exchanging messages through the communication channels. Each node has its own local state which is characterized it's initial state and operation history [1]. An enormous number of nodes can exist in the (large-scale) system however the number of nodes is assumed to be finite. This implies that a distributed algorithm which requires the communication spread over all the nodes in the system is not practical.

We assume the distributed systems are dynamic [3], where nodes can frequently join and leave the system. This means that the network's topology is changeable and each node never recognizes the entire system's configurations.

Each node has a comparable unique identifier (ID) drawn from a totally ordered set. The system is a fully connected network where all the nodes can directly communicate with all other nodes. All links used for the message transfer are reliable and FIFO, so all the messages are eventually received and the messages sent using the same link (in the same direction) are received in the order they were sent. We assume an asynchronous distributed system, where messages experience finite but unpredictable delay.

We also assume that distributed systems considered in this Chapter are applicable to checkpointing-rollback paradigm. This means that the distributed systems can restart executions from the restored snapshots, and thus we exclude the distributed systems with real-time interactions or human-interactions.

2.3.2 SSS Algorithm

Our work, CSS algorithm, is based on SSS algorithm that creates a partial snapshot of a subsystem [41, 42]. We briefly explain SSS algorithm before introducing CSS algorithm (Section 3.4). SSS algorithm has the following properties. (a) Snapshots are only created among communication-related nodes so that a rollback can only be done for the nodes related to a crashed node. Therefore, even if there are plenty of nodes in a distributed system, SSS algorithm can efficiently make a partial snapshot, and effectively achieve a rollback. (b) SSS algorithm is applicable to

dynamic distributed systems (where nodes can join and leave freely), because each node doesn't have to know the set of nodes in the system beforehand. Since a set of nodes contained in a partial snapshot is dynamically determined during the execution of SSS algorithm, it can cope with dynamic joining and leaving of nodes.

To achieve the above properties, SSS algorithm traces the communication-relation of the application algorithm: when a node records its checkpoint, it only requests nodes with which it has communicated to record their checkpoints. To realize this strategy, each node $node_i$ maintains a set DS_i, called a dependency set, of nodes with which it has communicated (sent or received messages). The set determines neighbors of the subsystem for which a partial snapshot should be taken. When a node $node_i$ receives an application message from another node, say $node_j$, or sends one to $node_j$, then the ID of $node_j$ is added to DS_i. DS_i always includes its own ID, $node_i$.

When a node $node_i$ initiates SSS algorithm, it records its local state and sends a marker with its ID to the nodes in DS_i at that moment. This node is called an *initiator*. When a node $node_j$, which is not an initiator, receives a marker for the first time, it records its local state and forwards the marker to each $node_x$ in DS_i. By this way, the marker is forwarded to all nodes in the subsytem for which a partial snapshot should be taken. On receipt the marker, $node_j$ sends a pair of its DS contents and ID $(DS_j, node_j)$ to the initiator, which maintains the union of the received dependency sets (including its own) and the set of received IDs (including its own). When the union of the dependency sets and the set of IDs become equal, the initiator decides that the nodes in the set constitute the subsystem where the partial snapshot should be taken. We call this a *partial snapshot group* (dependency relation group, DRG). Based on this set, the initiator informs each node $node_j$ of the set DRG_j of nodes from which $node_j$ should receive markers. This set can be defined as follows: $DRG_j = \{\ node_x \mid node_j \in DS_x\}$

Recording application messages and finishing the snapshot algorithm are done in the same way as traditional algorithms. Like as Chandy's algorithm, SSS algorithm allows the application algorithm to continue running during execution of the snapshot algorithm. To guarantee consistency, a node has to send a marker to another node that isn't contained in its DS before sending an application message. When SSS algorithm is terminated, each node in DRG clears its DS. With these procedures, SSS algorithm efficiently allows a small portion of a consistent snapshot efficiently. Thus, SSS algorithm can make the checkpoint correctly, even in a dynamic distributed system where the network topology is never statically fixed. However, as stated in Section 3.2, SSS algorithm does not guarantee consistency if there is a collision in a node. Our proposed algorithm solves this problem without sacrificing efficiency.

2.3.3 Problem Definition

Moriya and Araragi [41, 42] assume that two or more SSS algorithms are NOT simultaneously executed on the same node to guarantee the consistency of the partial snapshot. However, in some distributed applications, each node may need to initiate the snapshot algorithm independently from other nodes in order to confirm its current local state and commit the result of the requests. If each node can initiate the snapshot algorithm (become an initiator) at any time, SSS algorithm hardly guarantees its consistency because two or more snapshot algorithms may be collided each other. A naive solution to such a problem is that an initiator of the snapshot algorithm informs every node in the system of its initiation (i.e. broadcasting). However, the solution is not practical, especially in a large-scale distributed system. We define a *collision of SSS algorithms on a node* to be a situation where two or more SSS algorithms are executed at the same moment on the node.

Now we explain the problem of a collision in SSS algorithm. Here we assume that two or more SSS algorithms are operated independently and correctly. And that, each node maintains only the latest checkpoint. Therefore, when each node concurrently engages in two or more executions of SSS algorithm, it will keep only the one of the checkpoints created by the executions. Figure 2.1 shows two different executions, where only the behaviors of two nodes $node_a$ and $node_b$ among many nodes in the system are illustrated. In the both executions, collisions occur at the two nodes between two SSS algorithm SS_1 and SS_2. First, we consider the case that each node maintains the checkpoint recorded by the execution that *started* the most recently (Fig. 2.1(a)). In this case, $node_a$ maintains $ckpt_{a2}$ of SS_2 and $node_b$ does $ckpt_{b2}$ of SS_1. Thus, if $node_a$ and $node_b$ rollback for recovery, $node_a$ returns to checkpoint $ckpt_{a2}$ and $node_b$ does to checkpoint $ckpt_{b2}$. Sending of message msg_{ab} is recorded in $ckpt_{a2}$, however, msg_{ab} is NOT recorded (even as an in-transit message) in $ckpt_{b2}$ since SS_1 takes a consistent partial snapshot and sending of msg_{ab} is not recorded in $ckpt_{a1}$. This violates the strong consistency of the snapshot algorithm as mentioned in Section 3.1. Second, we consider the case that each node maintains the checkpoint recorded by the execution that *terminated* the most recently (Fig. 2.1(b)). In this case, $node_a$ maintains $ckpt_{a1}$ and $node_b$ does $ckpt_{b2}$. Because SS_1 is terminated at $ckpt_{a4}$ in $node_a$ and SS_2 is terminated at $ckpt_{b4}$ in $node_b$. By the observation similar to the first case, $ckpt_{b2}$ records the receipt of msg_{ab} but $ckpt_{a1}$ does not record the sending of msg_{ab}. This causes msg_{ab} to be an orphan, which violates the consistency of the snapshot.

To guarantee the consistency, one possible approach is to modify the rollback algorithm so that the nodes do not necessarily return to their last checkpoints but return to some adequate

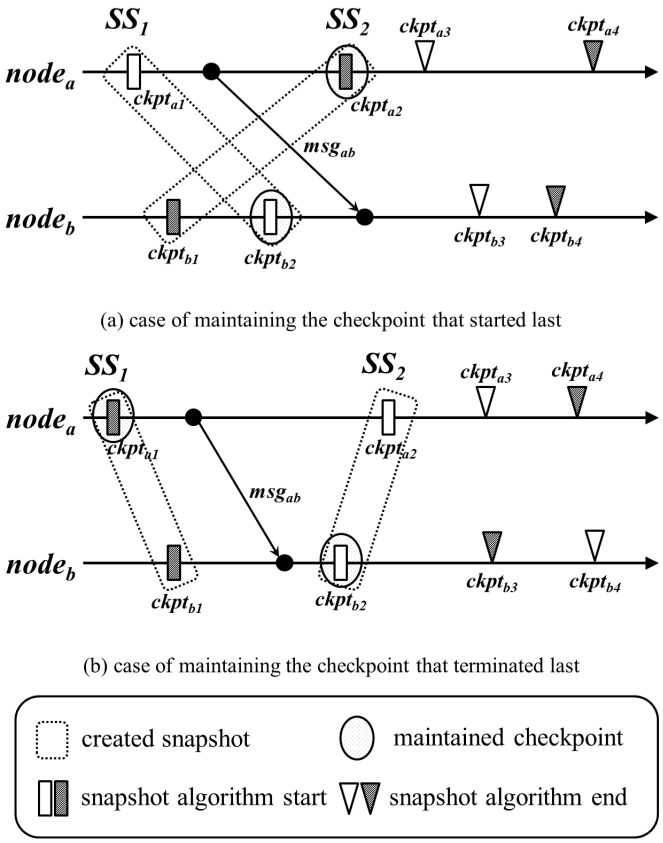


Figure 2.1: Concurrent executions of SSS algorithm

checkpoints to avoid such inconsistency. However, we need a sophisticated method for finding adequate checkpoints. In addition, each node has to keep multiple checkpoints, up to the number of nodes, while Chandy's algorithm and SSS algorithm require each node to store only the latest checkpoint it takes. Thus, we lose the efficiency achieved by Chandy's algorithm and SSS algorithm. In our works, we take another approach. We modify SSS algorithm so that concurrent SSS algorithm executions in collision are combined to a single SSS algorithm execution. With this merging, recording the latest checkpoint at each process is sufficient and the original efficient rollback procedure can be utilized. As we mentioned previously, we call our partial snapshot algorithm *Concurrent Sub-Snapshot (CSS)* algorithm.

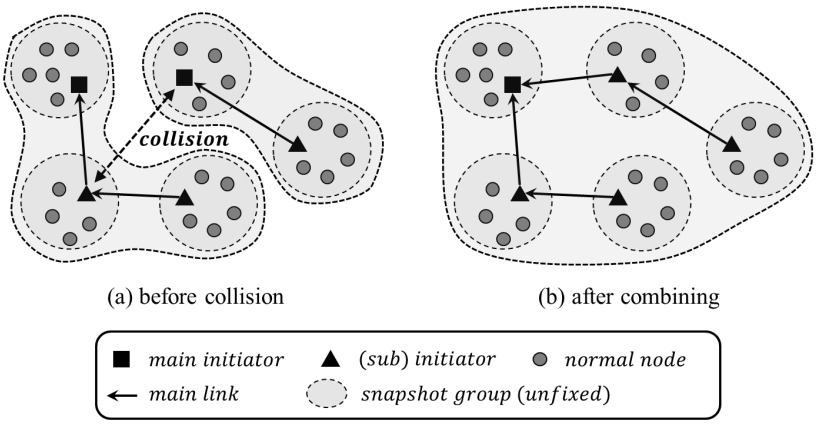


Figure 2.2: Combining of two executions of SSS algorithm

2.4 Concurrent Partial Snapshot Algorithm

In this section, we introduce CSS algorithm that can combine the concurrently executed SSS algorithms when a collision occurs.

2.4.1 Overview of CSS Algorithm

CSS algorithm solves the collision problem by combining two or more colliding SSS algorithms. In the CSS algorithm, a representative node, called a *main-initiator*, is selected among the initiators of the colliding SSS algorithms. Then the main-initiator behaves as an initiator of the combined SSS algorithms.

Fig. 2.2(a) shows two partial snapshot groups concurrently executing CSS algorithm. Each group includes a unique initiator and consists of nodes communication-related to the initiator and each snapshot group is NOT fixed currently. If there is a communication-relation between two nodes that belong to different partial snapshot groups, the collision will occur between these two groups. Note that either this communication-relation can be created after initiating the snapshot algorithm, or this communication-relation can already exist in the same snapshot group that is initiated by two or more different initiators. The node that is related to another group's node recognizes the collision when receiving a new marker with different initiator ID. The node that detects the collision informs its group's initiator (this informing message will be forwarded to main-initiator of the group). If the initiator has not determined its partial snapshot group yet, the two executions of the SSS algorithm are combined. The initiator directly communicates with

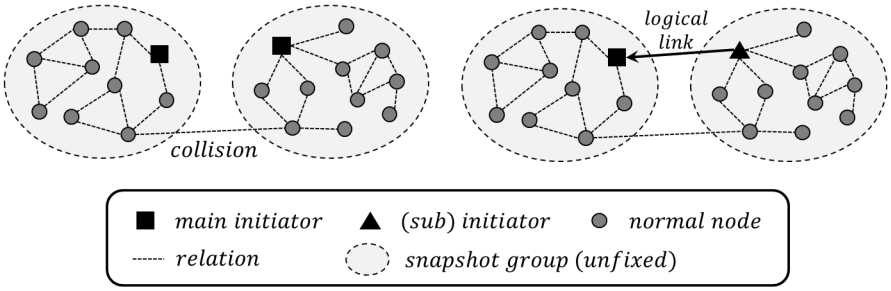


Figure 2.3: Combining of two executions of SSS algorithm

the other initiator to decide which initiator should be the main initiator of the combined SSS algorithm: the initiator with a prior ID becomes the main initiator. The other initiator is called a *sub-initiator*. The sub-initiator generates a logical link to the main-initiator that is called a *main link*. This process is depicted on Fig. 2.2(b). After the selection of the main-initiator, the sub-initiator passes the information (DSs and node IDs) it has collected and forwards incoming information to the main-initiator.

If another collision occurs between the combined SSS algorithm and another SSS algorithm, the combining is performed exactly in the same way, as depicted in Fig. 2.3. Note that the local state to be stored as a checkpoint of a collided node is the one at the time when the node received a marker for the first time. This way of storing achieves consistency and avoids deadlock. In the selection of the main-initiator, an additional procedure is performed (not mentioned here) to avoid deadlock. The details are explained in Section 3.4.3.

Our previous work, SSS algorithm, guarantees the efficiency of taking snapshot even if the system is large-scale, because SSS algorithm should be executed on the small subsystem consisting of communication-related nodes. This implies that the complexity of the snapshot algorithm is independent from the scale (the number of nodes) of the whole system. CSS algorithm, based on SSS algorithm, also preserves this property obviously. However, even the scale of the subsystem is same, higher efficiency might be expected by CSS algorithm because it can take the snapshot of the subsystem concurrently with two or more initiators.

2.4.2 CSS Algorithm

In this section, we give the pseudo codes of CSS algorithm.

Variables of each node

init : Stores an initiator's ID. The value is initially $\perp$. When a node receives a marker and the value of *init* is $\perp$, then it is set to the initiator ID of the marker. When the execution of CSS algorithm on the node terminates, init is initialized to $\perp$ (for the next execution of CSS algorithm).

MainLink : Stores the main-initiator's ID. The value is initially $\perp$. When the node initiates CSS algorithm, then the value is set to its own ID. When the node becomes a sub-initiator after it combines with another execution of CSS algorithm, the MainLink is set to the ID of the main-initiator.

WaitFlag : A Flag variable to consistently select a main-initiator when the collision occurs.

RcvMkList : This set variable stores the IDs of the nodes from which the node received markers.

DS : This set variable stores the IDs of the nodes in its Dependency Set.

DSSender : This set variable is used only by an initiator to store the IDs of the nodes from which DS information was received.

MustRcv : This set variable stores the IDs of the nodes from which the node has to receive markers (before termination).

AllDS : This set variable is used only by an initiator to store the union of the DSs it received.

MsgQ : This message queue stores the messages received during the process to resolve the collision. The stored messages are processed after the collision is resolved.

FinFlag : A flag variable to determine termination. Value will be TRUE when Fin message is received.

msg : The set variable to store currently received messages. This variable includes not only message's type but also attached parameters and sender's ID.

We use one constant variable, **self**, which stores each node's own ID.

Messages: Messages for CSS algorithm have the form of (< *Message_type*, Parameters>, Sender). The message types are listed below. The Sender is the ID of the sender. We introduce the following three types of messages for taking a snapshot and an additional five types of system messages for handling a collision.

<**Marker, Initiator**> Marker message. Parameter *Initiator* denotes the initiator's ID.

<**DSinfo, localDS, DSfrom**> A message to send its own DS (all the nodes communication-related to this node) to the initiator designated in the marker.

<**Fin, DRGinfo**> Termination message. Parameter $DRGinfo$ is the node list from which each node has to receive markers (before termination).

<**NewInit, CollidedNode, 2ndInit**> A informing message of detecting a different snapshot algorithm. The message is delivered to the main-initiator from a collided node (or the node detecting a collision). $CollidedNode$ is the collided node's ID, and $2ndInit$ is the ID of the initiator that caused the collision (the ID is contained in the marker that caused the collision). If the initiator (the destination of $NewInit$ message) has already become a sub-initiator, the message is relayed to the main-initiator using $MainLink$.

<**Accept, Initiator, 2ndInit**> A message which implies combining is possible. When a main-initiator receives a $NewInit$ message and decides that these two SSS algorithms should be combined, it sends *Accept* message to the collided node.

<**Combine, CollidedNode, 1stInit**> A informing message of combining two different snapshot algorithms. When a collided node receives an *Accept* message, it sends *Combine* message to the node that sent $Marker$ message that caused the collision. In the same way as $NewInit$ message, if the destination has already become a sub-initiator, it relays the message to the main-initiator using $MainLink$.

<**CompInit**> A message for compare the priority between two initiators' ID. When a main-initiator receives *Combine* message, it sends *CompInit* message directly to the other main-initiator to compare their IDs.

<**InitInfo, DSInfo, SenderInfo**> A message from a sub-initiator for notifying its information (including DS maintained) of a main-initiator. When two algorithms are combined, the main-initiator with low priority (now becomes a sub-initiator) sends $InitInfo$ message to forward the DS information and the DS senders' list it collected to the main-initiator with high priority.

Algorithm 1 to 10 represent the pseudo codes of CSS algorithm. Fig. 2.4 shows the flow of the messages from $Marker$ to *CompInit* when a collision occurs between two different executions of CSS algorithm.

Algorithm 1 CSS Protocol: Initialize Snapshot Algorithm

```
1: procedure INITIALIZE()
2:     MainLink ← self
3:     send(self, <Marker, self>)                ▷ send Marker to itself
4: end procedure
```

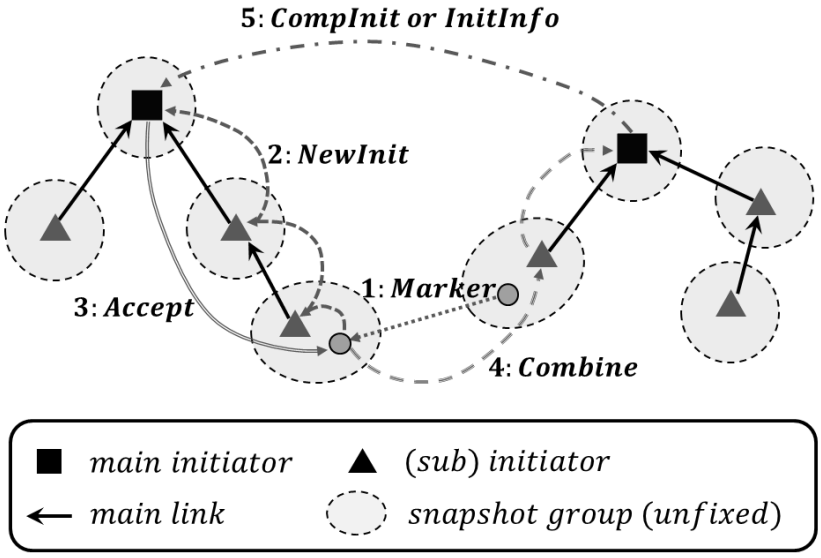


Figure 2.4: Message flow for handling a collision in CSS algorithm

Algorithm 2 CSS Protocol: When $Marker$ is received

```
5:  procedure OnReceive(<Marker, ?init>, ?sender)
6:      if init = ⊥ then                                  ▷ receipt of the first Marker
7:          Store Local State
8:          init ← ?init
9:          RcvMkList ← RcvMkList ∪ { ?sender }
10:         send(?init, < DSinfo, DS>)                    ▷ send current DS to the initiator
11:         send(∀node_i ∈ DS, <Marker, ?init>)           ▷ broadcast Marker
12:     else if init = ?init then                         ▷ second or later Marker
13:         RcvMkList ← RcvMkList ∪ { ?sender }
14:         if FinFlag then
15:             TerminationCheck()
16:         end if
17:     else                                              ▷ collision has occurred
18:         MsgQ.enqueue(msg)                             ▷ store Marker to the message queue
19:         send(init, <NewInit, self, ?init>)            ▷ notify the initiator of collision
20:     end if
21: end procedure
```

Algorithm 3 CSS Protocol: When $DSinfo$ is received

```
22: procedure OnReceive(<DSinfo, ?localDS, ?DSfrom>, ?sender )
23:     if MainLink = self then                          ▷ case of the main-initiator
24:         AllDS ← (AllDS ∪ ?localDS)                   ▷ to update group node list
25:         DSSender ← DSSender ∪ ?DSfrom                ▷ record sender node's ID
26:         if !WaitFlag then
27:             TerminationCheck()                       ▷ cannot terminate during combining
28:         end if
29:     else                                             ▷ message forwarding to the main-initiator
30:         send(MainLink, <DSinfo, ?localDS, ?DSfrom>)
31:     end if
32: end procedure
```

Algorithm 4 CSS Protocol: When Fin is received

```
33: procedure OnReceive(<Fin, ?DRGinfo>, ?sender)
34:     MustRcv ← DRGinfo
35:     FinFlag ← TRUE                                   ▷ FinFlag is ON
36:     TerminationCheck()
37: end procedure
```

Algorithm 5 CSS Protocol: When $NewInit$ is received

```
38: procedure OnReceive(<NewInit, ?node_i, ?init>, ?sender)
39:     if MainLink = self then                          ▷ case of main-initiator
40:         if WaitFlag then                             ▷ waiting condition
41:             MsgQ.enqueue(msg)
42:         else if DSSender ⊂ AllDS then                ▷ group is not decided yet
43:             send(?node_i, <Accept, self, ?init>)     ▷ permit for combine
44:             DS ← ( DS ∪ ?init )
45:             WaitFlag ← TRUE
46:         end if
47:     else                                             ▷ message forwarding to the main-initiator
48:         send(MainLink, <NewInit, ?node_i, ?init>)
49:     end if
50: end procedure
```

Algorithm 6 CSS Protocol: When *Accept* is received

```
51: procedure OnReceive(<Accept, ?init_i, ?init_j >, ?sender)
52:
53:     receive( MsgQ.Dequeue( msg(<Marker, ?init_q >, ?node_q ) )
54:     send(?init_j, <Combine, self, ?init_i > );
55:                                  ▷ notifying collision and combining to opponent initiator
56:     send(?node_q, <Marker, ?init_q > )                                        ▷ reply marker
57: end procedure
```

Algorithm 7 CSS Protocol: When *Combine* is received

```
58: procedure OnReceive(<Combine, ?node_k, ?init>, ?sender)
59:     if MainLink = self then                                  ▷ case of the main-initiator
60:         if self < ?init then                          ▷ self is prior to the opponent initiator
61:                                  ▷ WaitFlag is ignored to resolve the deadlock problem
62:             send(?init, <CompInit> )                ▷ the opponent will become sub-initiator
63:         else                                          ▷ opponent initiator is prior to self
64:             if WaitFlag then                                              ▷ WaitFlag is ON
65:                 MsgQ.Enqueue(msg)                 ▷ wait for the previous combine process
66:             else                                                ▷ case of being sub-initiator
67:                 send(?init, <InitInfo, AllDS, DSSender>)
68:                 MainLink ← ?init
69:             end if
70:         end if
71:     else                                          ▷ forwarding message to MainLink
72:         send( MainLink, <Combine, ?node_k, ?init> )
73:     end if
74: end procedure
```

Algorithm 8 CSS Protocol: When *CompInit* is received

```
75: procedure OnReceive(<CompInit>, ?sender )
76:     send( ?sender, <InitInfo, AllDS, DSSender> )
77:     MainLink ← ?sender
78:     WaitFlag ← FALSE
79:     Process the messages in MsgQ
80: end procedure
```

Algorithm 9 CSS Protocol: When $InitInfo$ is received

```
81: procedure OnReceive(<InitInfo, ?AllDS, ?DSSender>, ?sender)
82:     AllDS ← (AllDS ∪ ?AllDS)
83:     DSSender ← (DSSender ∪ ?DSSender)
84:                          ▷ apply opponent's DS and Senders to maintain combined group
85:     WaitFlag ← FALSE
86:     TerminationCheck()
87: end procedure
```

Algorithm 10 CSS Protocol: Termination Check Method

```
88: procedure TerminationCheck()
89:     if MainLink=self then                          ▷ case of the main-initiator
90:         if AllDS ⊆ DSSender then
91:             send(∀node_i ∈ AllDS, <Fin, {node_j | node_j sent the marker to node_i} >)
92:         end if
93:     else                                  ▷ all other nodes except main-initiator
94:         if MustRcv ⊆ RcvMkList then
95:             terminate algorithm
96:         end if
97:     end if
98: end procedure
```

In pseudo codes, lines 1 to 3 represent the initiation of the snapshot algorithm. Each node can be an initiator by sending $Marker$ to itself. Note that procedure $Initialize()$ can be executed at any time by any nodes.

Lines 4 to 20 show the procedure executed when the $Marker$ is received. If the node which

received the $Marker$ has no initiator (line 6), it stores its local state to stable storage and broadcasts $Marker$ messages to its communication-related nodes. After broadcasting, the node should receive the replying $Markers$ of broadcasted $Markers$ (line 12). These replying $Markers$ will be recorded (line 13) to determine termination of the snapshot algorithm.

Collision of two snapshot algorithms is found when the node received $Marker$ with an attached different initiator ID (line 17). The node which detects the collision notifies its original initiator (ID stored in variable *init*) of collision and opponent initiator's ID by $NewInit$ message (line 19). This $NewInit$ message will be forwarded through $MainLink$ (line 48) if the initiator received the $NewInit$ message is not a main-initiator. The main-initiator which received $NewInit$ message checks whether DRG is fixed or not (line 42). Remember that DS_i includes $node_i$ itself (mentioned in Section 3.3.2), thus DSSender is proper subset of the AllDS unless all nodes in AllDS send their DS to the initiator. If DRG is not fixed, the initiator sends *Accept* message to combine the two collided snapshot algorithms. Note that the opponent initiator never terminates its snapshot algorithm because the node which detects the collision didn't send any replies to received markers. Moreover, the initiator which sends *Accept* message to the opponent initiator cannot terminate the snapshot algorithm without receipt of a replying message of *Accept* message($CompInit$ or $InitInfo$ message) because it adds the opponent initiator's ID to its DS (line 44) and $WaitFlag$ is true (line 45).

The node that received *Accept* message sends $Combine$ message to the $Marker$'s sender (lines 51 to 57). Same as $NewInit$ message, $Combine$ message will be forwarded through $MainLink$ (line 72) if the initiator that received the $Combine$ message is not a main-initiator. Unless $WaitFlag$ is true, the main-initiator compares its own ID's priority and the opponent's priority. If its ID is prior to the opponent's ID, $CompInit$ message should be sent. If not, $InitInfo$ should be sent and its $MainLink$ should be updated to the opponent's ID. However, even though $WaitFlag$ is true, if its own ID is prior to the opponent ID (line 60), it sends $CompInit$ message to the opponent to avoid deadlock (details will be mentioned in Section 3.4.3).

Procedure $TerminationCheck()$ (line 88 to 98) should be executed for checking termination of the snapshot algorithm. After terminating algorithm (line 95), all variables will be initiated to the initial values and DS information before initiating snapshot algorithm will be cleared.

2.4.3 Technical Discussion of CSS Algorithm

In this section, we give proof sketches for the correctness of CSS algorithm. First, we show that it will eventually terminate without deadlock. Next, we prove that the partial snapshot obtained

by it is consistent.

Deadlock Problem and Solution

In CSS algorithm, node IDs are compared among main-initiators to decide a new main-initiator for the combined partial snapshot. However, if two combining procedures occur on a main-initiator in parallel, and the initiator becomes a sub-initiator for one combining before the *Combine* message reaches the main-initiator of the other, then it causes an inconsistent situation. To prevent this problem, the main-initiator that permits combining must not become a sub-initiator by another CSS algorithm until the *CompInit* message is received. However, the introduction of this waiting condition can cause a deadlock. For instance, in Fig. 5, three main-initiators are waiting for replies from each other. To solve this deadlock problem, CSS algorithm uses an exception handling of a waiting condition. Even if a main-initiator is in a waiting condition, it replies with the *CompInit* message to the opponent main-initiator if the opponent's ID priority is lower than itself.

In Fig. 2.5, three main-initiators enter the waiting state and wait one another. In this case, at least one main-initiator has a higher priority than the its waiting main-initiators. For example, let the main-initiators of groups 1, 2, and 3 have IDs α, β, and γ respectively, where the priority level is $\alpha > \beta > \gamma$. In this case, initiator α receives the *Combine* message from initiator γ and initiator α learns that initiator γ should become a sub-initiator after combining. After initiator α sends CompInit to initiator γ, initiator γ leaves its waiting condition and the deadlock is released. In this manner, if such a waiting state occurs, at least one node eventually receives *Combine* message from a main-initiator with a lower priority. Therefore the deadlock is avoided by this exception

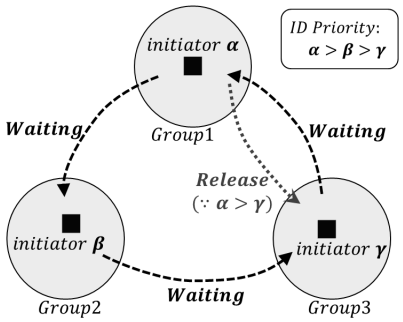


Figure 2.5: Example of deadlock

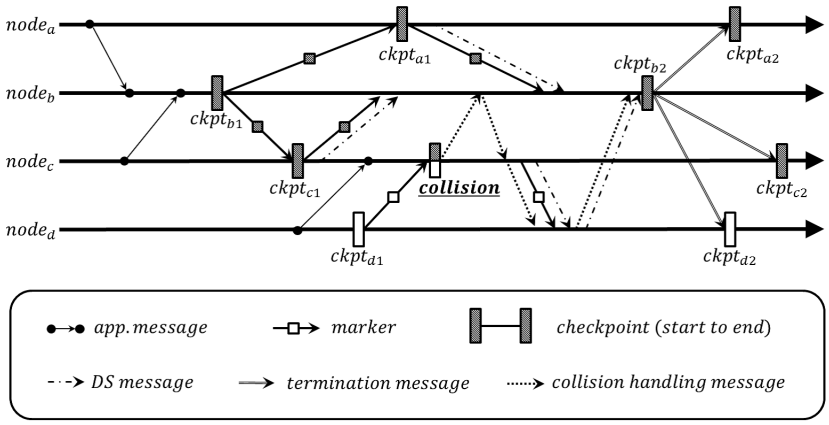


Figure 2.6: Execution example of CSS algorithm

handling operation.

Snapshot Consistency

Theorem 1. *The combined partial snapshot obtained by CSS algorithm is consistent.*

Proof sketch. We show that any partial snapshot obtained by CSS algorithm can also be obtained by SSS algorithm executed for distributed systems running the same application. Because the consistency of the partial snapshot by SSS algorithm has already been proven in [41, 42], we deduce that the partial snapshot by CSS algorithm is consistent. For any given execution of CSS algorithm, we can construct an execution of SSS algorithm that obtains the same partial snapshot. To help understanding, we show a construction example that deals with the messages employed in partial snapshots: Application messages, markers, dependent set information (DS), termination messages and system messages for handling the collision.

Consider the execution of CSS algorithm in Fig. 2.6. Each $node_i$ starts a snapshot algorithm at $ckpt_{i1}$ and terminates taking a checkpoint at $ckpt_{i2}$. In this execution, two partial snapshot algorithms are individually initiated at $ckpt_{b1}$ of $node_b$ and $ckpt_{d1}$ of $node_d$. They collide on $node_c$ and are combined. In the combining, $node_b$ becomes a main-initiator because of its higher priority on the node's ID. Each $node_i$ except for the initiators begins the snapshot algorithm and stores its local state when it receives the marker for the first time at $ckpt_{i1}$. The main-initiator terminates the partial snapshot algorithm when it broadcasts termination messages, and each $node_i$, except the main-initiator $node_b$, terminates when it receives the termination message and receives all the markers that the node has to receive. The termination messages include the node

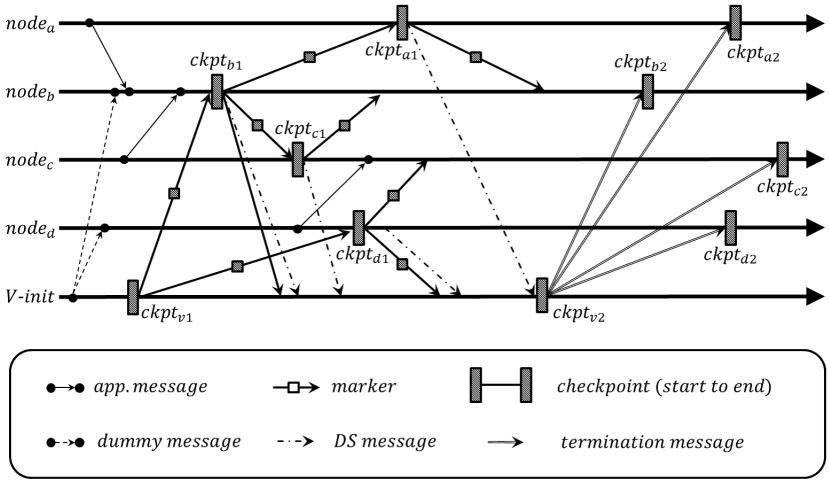


Figure 2.7: SSS algorithm simulation

list and each $node_i$ has to receive the markers from the nodes in the node list. Then, each node can determine which message links should be recorded, where the messages received between $ckpt_{i1}$ and $ckpt_{i2}$ are stored as in-transit messages. Each node must store the message link state in its checkpoint to achieve consistency.

Next we explain how the corresponding execution (the SSS simulation) of SSS algorithm is constructed from a given CSS algorithm execution. The SSS simulation has the same checkpoints as an original execution of CSS algorithm, while keeping the dependency of the application messages. Fig. 2.7 shows the SSS simulation of the CSS algorithm execution in Fig. 2.6. We assume that the transfer pattern of the application messages is the same at both the CSS execution and the SSS simulation to make identical system behavior. We show the construction by using this example. In the SSS simulation, a virtual initiator *V-Init* is introduced to the original distributed system. At any time, *V-Init* can send virtual application messages (*dummy messages*) that have no effect on the application (i.e., they are ignored when received). In the SSS simulation, *V-Init* sends two dummy messages to $node_b$ and $node_d$ before initiating the snapshot algorithm to establish a communication-relation. First, we assume that *V-init* initiates SSS algorithm at $ckpt_{v1}$ and sends markers to $node_b$ and $node_d$ because of the communication-relation created by the two dummy messages. In this simulation, markers from *V-init* can be received at the exact same timing with the CSS execution ($ckpt_{i1}$ in Fig. 6) because the message delivery can be delayed arbitrarily in the asynchronous distributed system. Therefore, the initiating time of

each node is the same as the CSS execution example. Next, in the CSS execution (Fig. 2.6), the markers initiated by $node_b$ and $node_d$ are replaced by the markers from *V-Init*. The initiator value contained in them is changed to *V-Init*. Therefore the destinations of the DS information messages are also changed to *V-Init*. The order of the arrivals of the DS information messages at *V-Init* can be decided arbitrarily. We assume that the snapshot group of the CSS algorithm execution of Fig. 2.6 is the group of all nodes. Accordingly, in the SSS simulation, *V-Init* terminates the partial snapshot algorithm when it receives DS information messages from all the nodes. In Figure 2.7, *V-Init* terminates the algorithm at $ckpt_{v2}$ and broadcasts the termination messages to all the other nodes. Finally, each node $node_i$ except *V-Init* terminates the algorithm at $ckpt_{i2}$ when it receives the termination message.

We show that the SSS simulation is a possible execution of SSS algorithm. The markers and the DS messages are clearly enabled at their sending, and don't contradict the event dependencies. In the SSS simulation, each termination message depends on all of the DS messages each of which depends on $ckpt_{i1}$. In the CSS execution of Fig. 6, the termination of the main initiator ($node_b$) depends on all of the DS messages after the communication of the collision-handling messages, and the termination of the nodes other than $node_b$ depends on its termination. On the other hand, the DS information message of each $node_i$ depends on $ckpt_{i1}$. In the CSS execution, each $ckpt_{i2}$ depends on all of $ckpt_{j1}$. Therefore, we assume that each $node_i$ receives the termination message at $ckpt_{i2}$ in the SSS simulation without producing any contradiction on event dependency. As those, the SSS simulation presents one possible execution of the SSS algorithm.

Finally, we show that the two partial snapshots are the same. To ensure that, we show the following three claims: (a) The set of nodes participating in the partial snapshot (snapshot group) by CSS algorithm is the same as that by SSS algorithm. (b) For any node, a local state recorded by CSS is the same as that by SSS. (c) For any node, the state of every message link recorded by CSS is the same as that by SSS. For each $node_i$, $ckpt_{i1}$ in the SSS simulation and $ckpt_{i1}$ in the CSS execution occur at the same time relative to the events of receiving application messages, and then the recorded local states are identical among them. This proves (b). Similarly, all corresponding DS information messages are the same between the two executions. In the SSS simulation, all the DS messages are received by *V-Init* and the order of receiving them can be decided arbitrarily, because it does not affect the event dependency. Then we can assume that the order is same as the order of receiving DS information messages by main initiator $node_b$ in the CSS execution. As a result, in the SSS simulation, *V-Init* decides the same snapshot group as $node_b$ does in the CSS execution. Then condition (a) is satisfied. Last, $ckpt_{i1}$ and $ckpt_{i2}$ of both executions occur at the same timing between the two executions relative to the events of receiving

application messages, and their snapshot groups are identical. Then the recorded message links and their contents are exactly the same between them. This implies condition (c).

2.5 Performance Evaluation

In this section, we present our experimental results to show the practical performance of CSS algorithm.

We evaluated our proposed algorithm with the following experiments; two kinds of latency experiments and one throughput experiment. In the first latency experiment, we compared the response times among the following three systems; one taking NO checkpoint, another taking checkpoints with SSS algorithm, and the other taking checkpoints with CSS algorithm. To evaluate the overhead of our snapshot algorithm, we made experiments with changing the frequency of sending requests and taking checkpoints. In the second latency experiment, we compared the response times between two systems; one taking checkpoints with SSS algorithm and the other with CSS algorithm. In this experiment, we controlled the collision ratio to check the effect of collisions on the response time. In the throughput experiment, we compared two systems: NO checkpoints and checkpoints with CSS algorithm.

Our experiment was made for two different scales of distributed systems: 12 PCs and 36 PCs to confirm the scalability of our system.

2.5.1 Experiment Environment

As we mentioned in the introduction, a partial snapshot algorithm works efficiently when it is applied to dynamic and large-scale distributed systems such as web-service [52] systems specified by W3C [51]. For the performance evaluation of our algorithm, we implemented a sample application of market place that is the standard benchmark Testbed for W3C's web-service platform [10].

The scales of experiment environment, 12 PCs and 36 PCs, might seem to be relatively small for a large-scale market place. However, the scales are enough for our evaluation because the communication of market place are done among (dynamically changing) independent small groups of nodes, and then the number of the nodes that joined each partial snapshot is relatively small. Moreover, if we increase the frequency of partial snapshots, the size of the groups becomes smaller. Actually, for those scales, partial snapshots seldom spread to the entire system in a reasonable frequency, for example, each timing that a task of a client is completed.

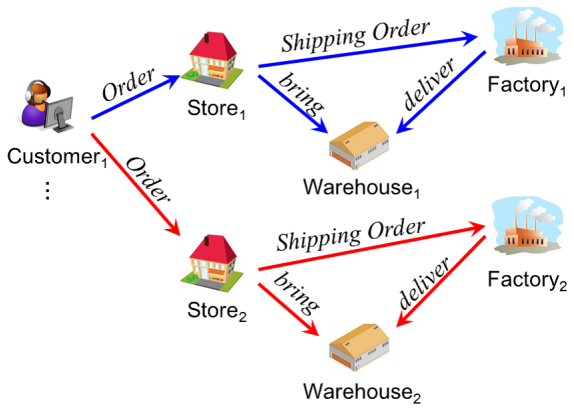


Figure 2.8: Structure of implemented web-service

For the experiments, we implemented a standard application of a web-service system: a shopping supply chain consisting of several service nodes (servers). In this application, customer nodes send requests to several store nodes. When a store node receives a request from a customer, it begins to process the received request and exchanges message between warehouse and factory nodes if necessary. Note that some nodes do not communicate with warehouse or factory nodes and this implies that the dependencies among nodes might be generated or not. The structure of our application is depicted in Fig. 2.8.

The system environment is shown in Table 3.1.

Table 2.1: Specifications of PCs

Number of PCs	12 or 36
CPU	Intel Core i3 2100 (3.1 GHz)
RAM	DDR3 4 GB
HDD	S-ATAII 500 GB
OS	CentOS 5.7
Platform	Apache Tomcat 6, Axis2 1.4.1

In the following, we show the setting of each experiment and the experimental results.

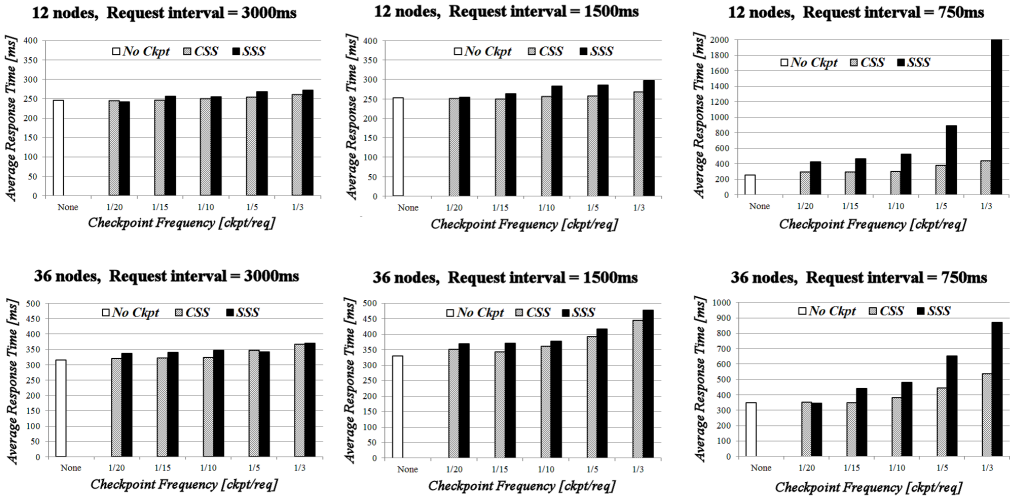


Figure 2.9: Result of latency experiments

2.5.2 Latency by request and checkpoint frequency

In this experiment, snapshot algorithms are initiated by two customers for every fixed number of requests that they have received. We call this interval a *checkpoint interval*. To control the frequency of requests, we set up a new node called a coordinator node. Requests are issued to customers in a round-robin fashion by the coordinator node at every fixed time interval. We call the interval at which requests arrive a customer *request interval*. For the system with SSS algorithm, an initiator can begin a new execution of SSS algorithm after the terminations of the snapshot algorithms initiated by other customers to avoid a collision. In both the systems with SSS and CSS, after its own initiation, the initiator has to wait for the termination before processing the next request to confirm the commitment of the previous requests. In the experiment, we measured the average response times of the requests by changing the frequency of the requests and the checkpoints whose frequency is controlled by the request and checkpoint interval. Of course, the more frequent checkpoints are taken, the less nodes are involved in each checkpoint because the extent of dependency among the nodes through communication has not much grown. After that, we measured the average response times of the requests. The result is shown in Fig. 2.9.

From this result, the difference of the latencies among the three systems is slight, except when the request and checkpoint intervals are short. This means that the overhead of CSS and SSS

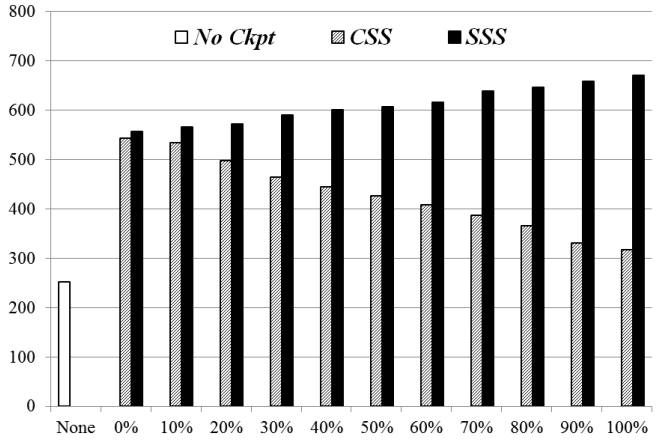


Figure 2.10: Average response time by changing collision ratios

algorithms for checkpointing can be ignored compared with the normal operation of applications. In the high frequency (high processing load) case, the collision ratio is increasing and the response of the system with SSS is delayed.

2.5.3 Latency by collision ratio

The basic setting of the experiment is the same as the first one. We fixed the checkpoint interval to every five requests and the request interval to 900 ms. For the system with CSS algorithm, we call the ratio of the occurring collisions between partial snapshots the *collision ratio*. In this experiment, we changed the collision ratio by manipulating the timing of initiating the checkpoint algorithm for one of the customers. For example, if the coordinator node sends the checkpoint request to an initiator just after the other node initiates the snapshot algorithm, they will collide. On the other hand, if the coordinator node sends the checkpoint request to an initiator after sufficiently long time to complete the checkpoint by the previous initiator, they will not collide. We changed the timing of sending the checkpoint requests to vary the collision ratio by holding specified checkpoint frequencies.

Figure 2.10 shows the average response times of both systems with CSS and SSS algorithms by changing collision ratios. The response time of SSS algorithm increased, but CSS decreased because the system with SSS algorithm and a high collision ratio has to wait for a long time until the previously scheduled checkpointings, and the processing of subsequent requests are blocked.

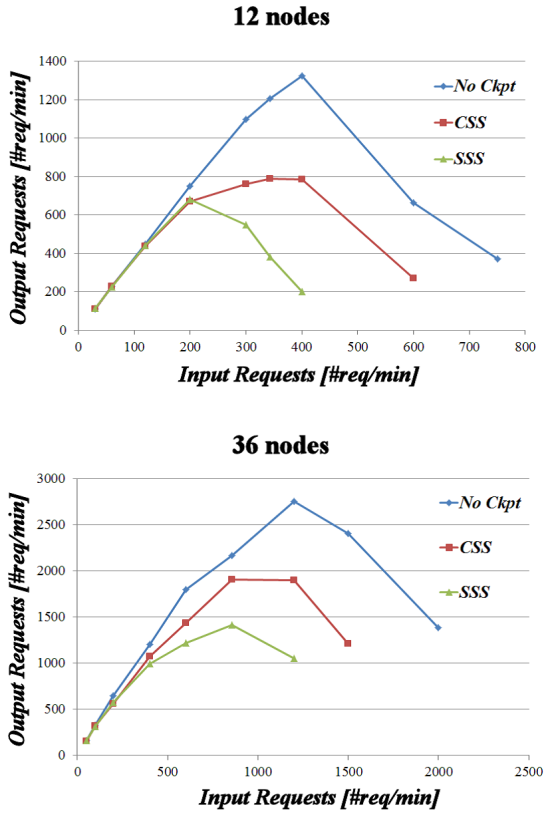


Figure 2.11: Result of throughput experiments

On the other hand, in the system with CSS, even if a collision occurs, the collided partial snapshots are combined. From a different point of view, the resulting snapshot is divided and processed in parallel. Therefore, we conclude that the advantage of speed up by this parallelization exceeds the merging overhead, and fast response is achieved.

2.5.4 Throughput

The setting of this experiment was the same as that of latency. We evaluated the processing capacity by this throughput experiment. The result is shown in Fig. 2.11. The extra consumption of resource by SSS and CSS algorithms is almost the same. However, after the resource limit, the throughput capacity of the system with SSS algorithm is exhausted faster. Because of the high collision ratio around the limit, requests are blocked and exceeding demands for resource consumption are accumulated.

2.6 Concluding Remarks

We proposed a concurrent partial snapshot algorithm CSS. This algorithm makes it feasible to take snapshots of large-scale and dynamic distributed systems characterized by the participation of a prodigious number of nodes. CSS algorithm can deal with situations called collisions in which two or more algorithms are simultaneously executing on a single node. Any node in a distributed system can initiate the partial snapshot algorithm. We implemented a virtual web-service and used it to evaluate CSS algorithm. In an environment with frequently overlapping algorithms, CSS algorithm operated efficiently. Therefore, we expect that it can efficiently support the fault-tolerance of large-scale dynamic distributed systems.

Chapter 3

A Cooperative NameNode Cluster for HDFS

3.1 Introduction

In this section, we represent a new architecture of Hadoop distributed file system (HDFS)[45, 26] which can guarantee the fault-tolerance using replicated servers (NameNodes).

3.1.1 Background

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"[18], which is made up of all digital data in the world. And they predict the size of digital universe will be 2.8ZB($2.8 * 10^{21}$ Bytes). We usually call these huge data, which are difficult (or impractical) to process with traditional data processing tools or applications, *Bigdata*[8]. 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[45, 44].

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 135TB 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)[26] and the other one is MapReduce[12]. HDFS is a distributed file system based on GFS (Google File System)[15], and MapReduce is a programming model for processing large data sets.

HDFS consists of only one coordinator (also known as a 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 coordinator[25, 23, 24].

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 [23, 22], in order to store 100 million files, a NameNode should have at least 60GB of RAM.

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

In this chapter, 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 a NameNodes fails, another NameNode takes it role immediately, instead of the failed NameNode.

- **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 [17, 30]. The definition and proof of weak-linearizability will be discussed in Section 3.5.

3.2 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)[48] 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)[49], 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 $\lfloor (N-1)/2 \rfloor$ crash faults of the JournalNodes, where N denotes the number of the JournalNodes. 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[11, 5] 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 [21] is proposed to resolve both the namespace limitation problem and the load balancing problem. Giraffa file system adopts $HBase$[46], which is the Hadoop distributed and scalable database and is the open source implementation of Google's BigTable[13]. 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.

3.3 Preliminaries

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 work.

3.3.1 HDFS(Hadoop Distributed File System)

HDFS[26] 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.

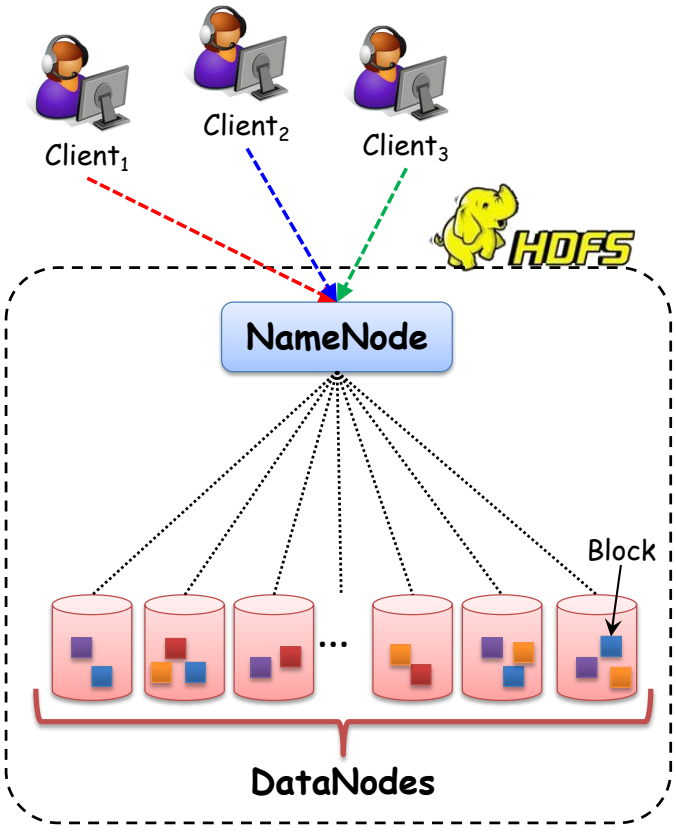


Figure 3.1: Architecture of HDFS

HDFS adopts a master/worker model. HDFS consists of one coordinator (master node) named NameNode, and many worker nodes named DataNodes (Fig. 3.1). 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.

3.3.2 ZooKeeper

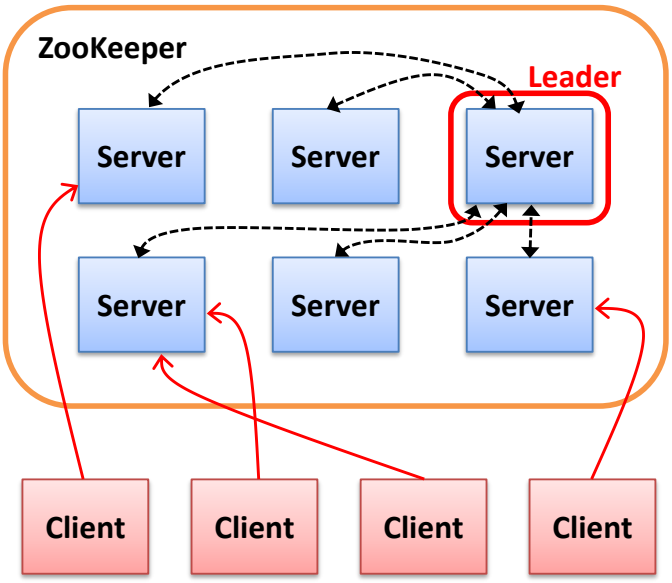


Figure 3.2: ZooKeeper Service

ZooKeeper[47] 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. 3.2). 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. 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.

3.3.3 Fault Model

In our model, 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.

3.4 Our approach: Cooperative Distirbuted NameNode Cluster

3.4.1 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.

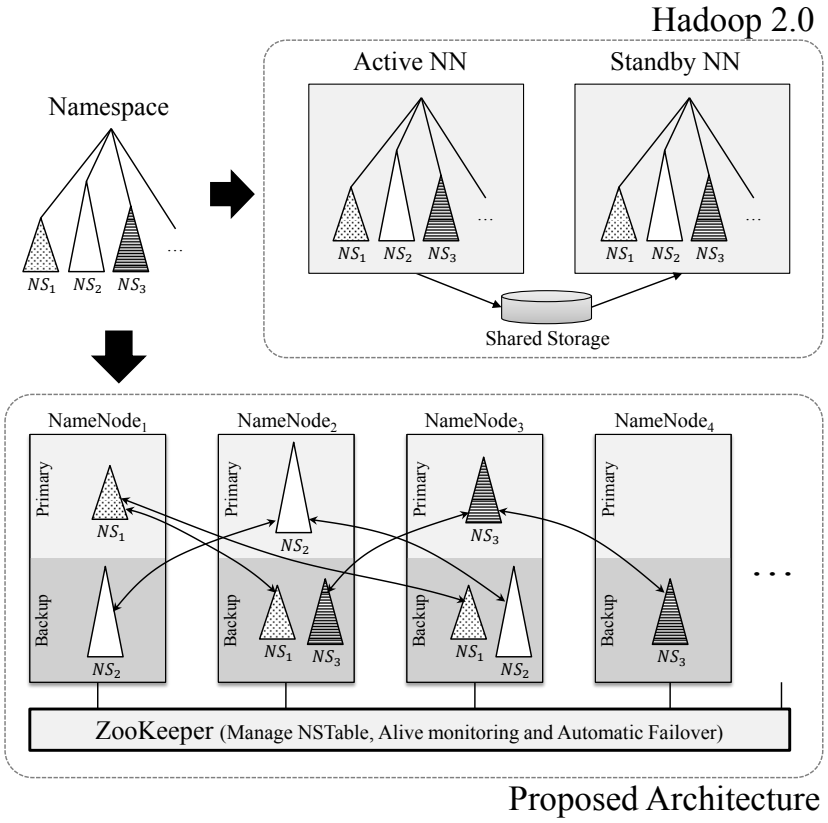


Figure 3.3: Namespace Partitioning

Our proposed system has several NameNodes. It partitions the namespace of HDFS and stores replicas of the fragments distributedly among the NameNodes. Figure 3.4.1 describes the difference between our system and Hadoop 2.0[48]. 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, ...

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

In contrast, the lower part of Fig. presents our proposed architecture. Our system contains N NameNodes denoted by $NameNode_1$, $NameNode_2$, ..., $NameNode_N$, and this set of NameNodes are called a **NameNode cluster**. The namespace is partitioned into m disjoint fragments(NS_1, NS_2, ..., NS_m). The fragments are replicated and distributedly stored in the NameNodes. In Fig. , 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. , 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 Section 3.4.2.

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.

3.4.2 Cooperative Process

In this Section, we explain how NameNodes cooperate.

Consistency Mechanisms

As mentioned in Section 3.4.1, 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. 3.4 shows the flow of processing a request from a client on the Primary NameNode. We

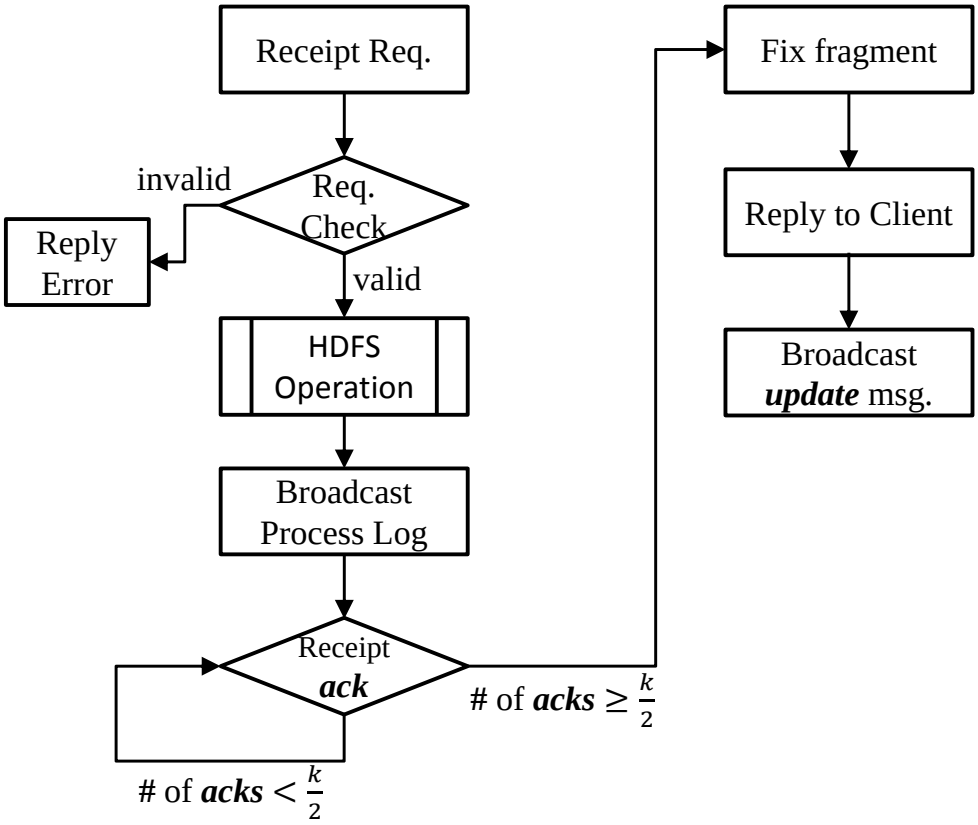


Figure 3.4: Process of the Request on a Primary NameNode

describe this process in detail through Algorithms from 11 to 13. 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.

Algorithm 11 NameNode Event: 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 12 NameNode Event: 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 13 NameNode Event: 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
```

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 creates 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 14 to 17.

Algorithm 14 NameNode Event: When *Suspect* from ZooKeeper is received

```
1: procedure OnSuspect(NN_x)
2:     Record its latest NS_{f_i}^{v_i} to znode on the ZooKeeper
3:     if self is the next PNN then
4:         Wait until a majority of the NS_f^v are recorded
5:         nextNS ← newest one among collected NS_f^vs
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 15 NameNode Event: 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 16 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 17 NameNode Event: 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
```

3.4.3 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 16).

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 16) 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 11 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.

3.5 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.

3.5.1 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 [17], 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**.

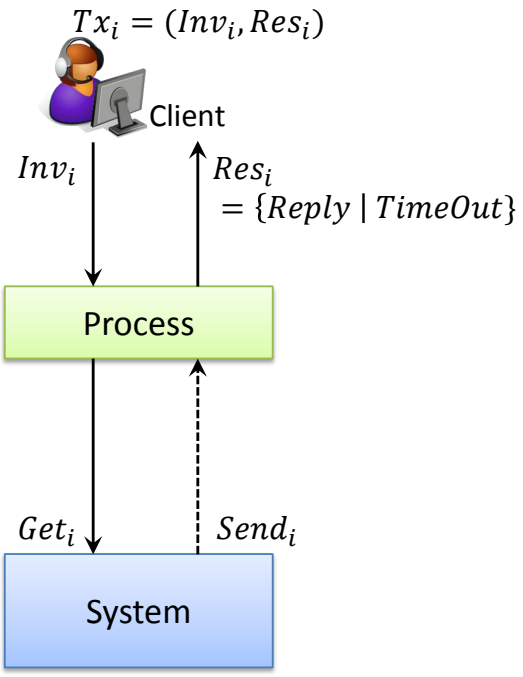


Figure 3.5: Representations of a Request

Fig. 3.5 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 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 $TimeOut$.

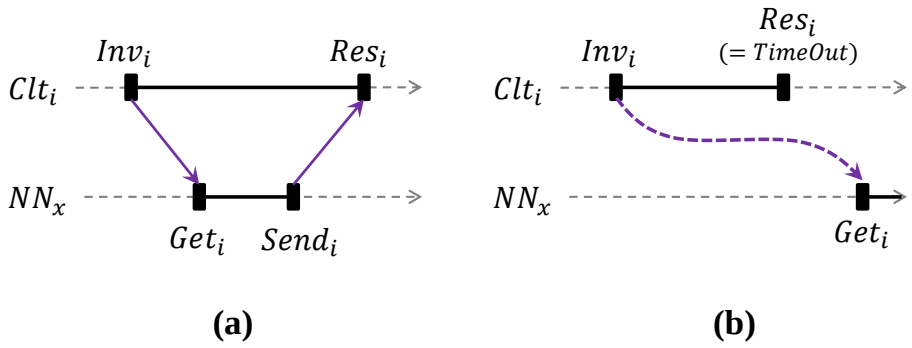


Figure 3.6: Timings of Inv and Res

Fig. 3.6 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. 3.6(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 $TimeOut$. Fig. 3.6(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 chapter, 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.

3.5.2 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: $\sigma_{NN_i} = [Req_i \prec Req_j \prec ...]$.*

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 $Consensus(\sigma_{NN_c})$ 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.*

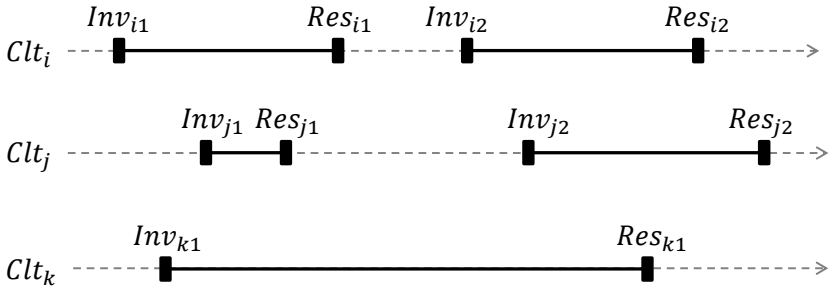


Figure 3.7: An Example of the Execution of 3 Clients

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. Fig. 3.7 shows an example of execution of 3 clients. In this case, we can construct the Global History $H_G = [Inv_{i1}, Inv_{k1}, Inv_{j1}, Res_{j1}, Res_{i1}, Inv_{i2}, ...]$.

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. 3.7, $H_G^{(Clt_j)}$ 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. $H_G^{(Clt_i)} = [Inv_0, Res_0, Inv_1, ..Res_n] = [Tx_0, Tx_1, ..Tx_n]$

Sequential Specification : A sequential specification consists of a set of operations and a set of sequences of operations[17]. 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 $H = [Inv_1, Res_1, Inv_2, Res_2, ...]$ has the result when the system processes the request $[Inv_1, Inv2, ..., Inv_i]$ 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[30, 17, 31]. 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.

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

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

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

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

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.

3.5.3 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 *Get*s 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) corre-

sponding 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:

$$\begin{aligned} Resync(NN_j) &\prec Sync(Get_j) \\ &\prec Resync(NN_i) \prec Sync(Get_i) \end{aligned}$$

$Resync()$ 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.

3.6 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.

3.6.1 Experimental Environments

We use 36 commodity servers for the experiments. Table 4.1 shows the specification of each server.

Table 3.1: 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)

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 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.

3.6.2 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).

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 100ms (10 requests per second). This frequency is very low and uninfluential to the system performance.

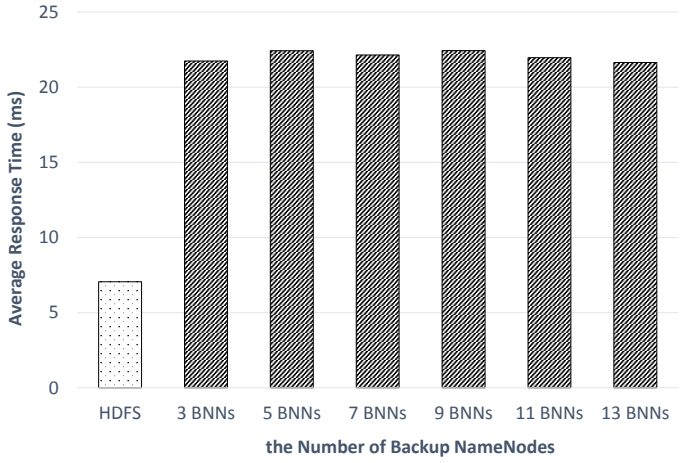


Figure 3.8: Overhead on Low Load

Fig. 3.8 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 7ms. However, our proposed system requires about 22ms, because of the synchronization overhead.

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.

Overhead on Heavy Load

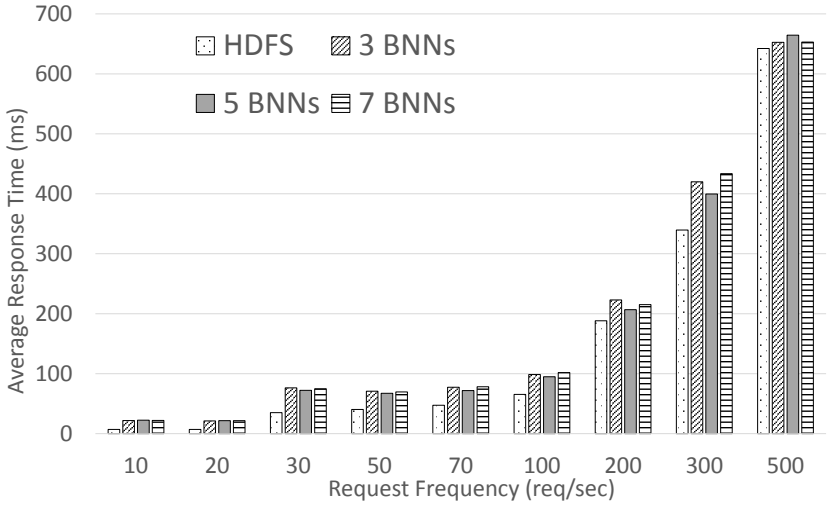
Now we change the frequency of the sending requests to evaluate average response time on various load environments.

Fig. 3.9 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. Fig. 3.9(a) shows the result on 3, 5, and 7 Backup NameNodes compared with the original HDFS, and Fig. 3.9(b) shows the result on 9, 11, and 13 Backup NameNodes.

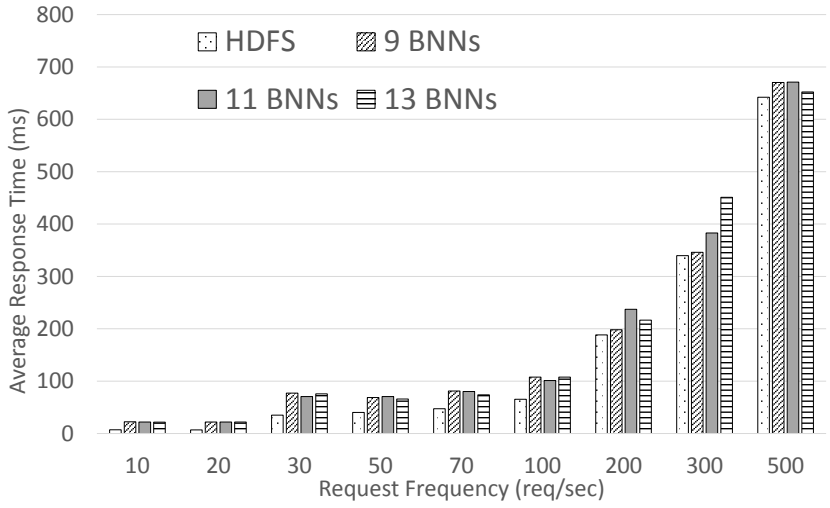
As we discussed in the previous section, the orignal HDFS's response time is shorter than our proposed system's when the system load is low. However, if the frequency of the requests exceeds 200 requests per second, the difference becomes smaller. In more detail, we found the average overhead of the synchronization is about 15ms in the case of low load. This overhead is relatively high because each request can be processed in about 7ms. However, as the system load increases, the synchronization overhead gradually increases up to about 50ms, 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 7ms to about 580ms. 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 Section 3.6.2, 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. Fig. 3.10 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 in-



(a)



(b)

Figure 3.9: Overhead on Heavy Load

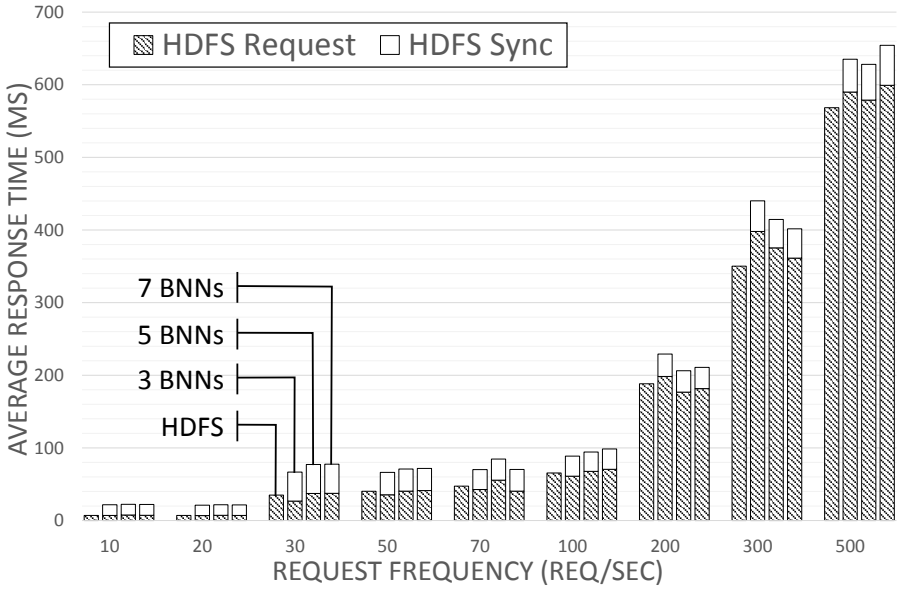


Figure 3.10: Ratio of Synchronization Overhead

creases 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.

3.6.3 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.

Overhead on Low Load

The same as Section 3.6.2(Low Load), 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.

Fig. 3.11 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 9ms, this result is only 2ms slower than the original HDFS. The optimization of the synchronization process can be cited as a possible

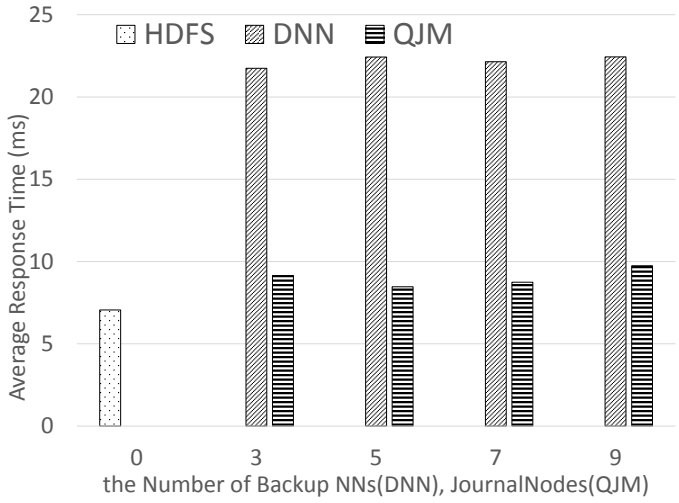


Figure 3.11: Comparing with QJM on Low Load

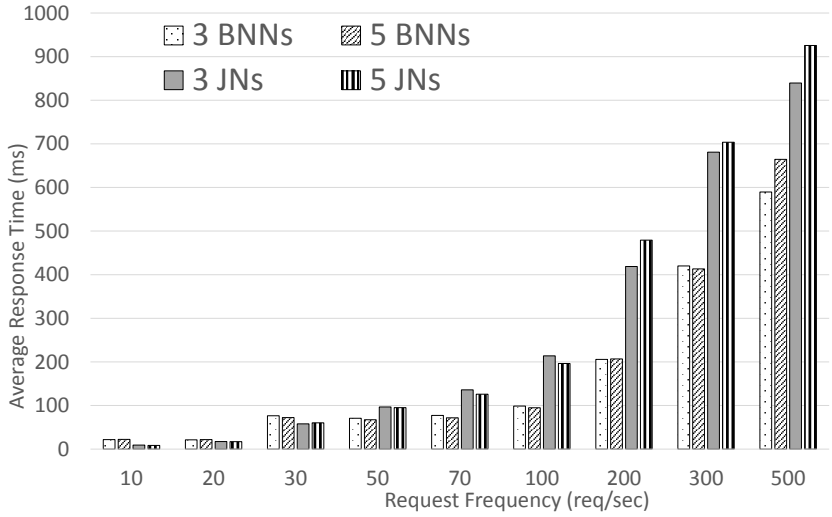
cause. QJM also hardly changes the average response time if the number of JournalNodes is increased.

Overhead on Heavy Load

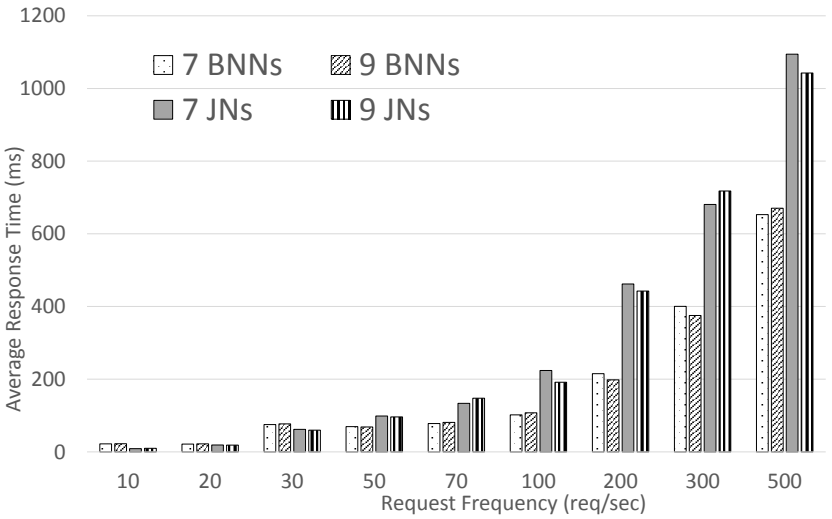
In this Section, we change the request frequency, and measure the average response time of our system and QJM.

Fig. 3.12 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. Fig. 3.12(a) shows the result on 3 and 5 Backup NameNodes, and Fig. 3.12(b) shows the result on 7 and 9 Backup NameNodes (JournalNodes in the case of QJM).

In the result in the Section 3.6.3(Low Load), 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.



(a)



(b)

Figure 3.12: Comparing with QJM on Heavy Load

3.6.4 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 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.

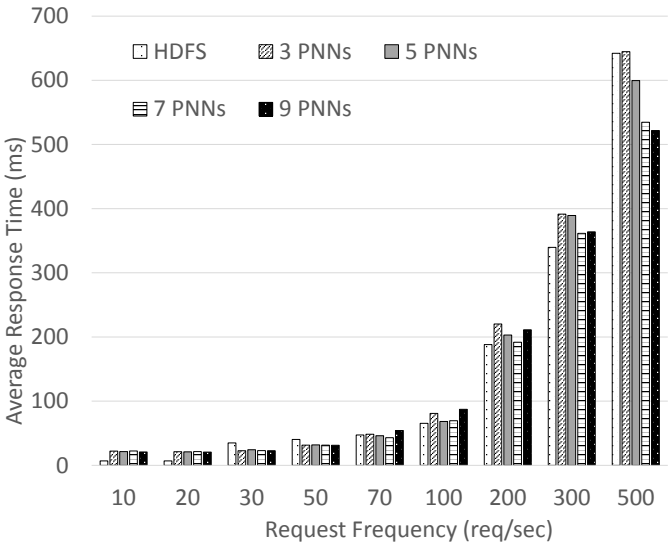


Figure 3.13: Effect of Load Balancing

Fig. 3.13 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.

3.6.5 Performance of MapReduce task

As we introduce in the Section 3.1, 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.

3.7 Concluding Remarks

In this chapter, 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 Section 3.6.5. 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.

Chapter 4

Conclusion

4.1 Summary of the Results

In this dissertation, we have considered the fault-tolerance of distributed systems. Fault-tolerance is one of the fundamental and important issues in distributed systems, because distributed systems are prone to fault. We proposed two methodologies of fault-tolerance using some practical examples in this dissertation.

In Chapter 2, our new snapshot algorithm, Concurrent Sub-Snapshot (CSS) algorithm, was introduced. A snapshot algorithm is the key technique for implementing the checkpoint-rollback recovery to guarantee fault-tolerance of the distributed system. Our snapshot algorithm is based on our previous work, Sub-SnapShot(SSS) algorithm, which takes a consistent global state of the subsystem. The same as SSS algorithm, CSS algorithm is also partial snapshot algorithm, thus its message complexity is independent on the entire system's scale. This implies that SSS algorithm and our snapshot algorithm are suited to the large-scale distributed systems. Moreover, we resolved the collision problem in SSS algorithm that can efficiently take a partial snapshot even when multiple nodes concurrently initiate the snapshot algorithms. When a collision occurs on a node, this means two or more snapshot domains are overlapped, CSS algorithm successfully realized an efficient solution for the collision problem by consistently merging two concurrent snapshot algorithms. And we implemented a virtual web-service to evaluate our proposed snapshot algorithm. From these evaluations, we showed that our snapshot algorithm operated efficiently, especially when the collision among snapshot algorithms are occurred frequently.

In Chapter 3, we proposed a new architecture of Hadoop distributed file system which uses multiple NameNodes. Our proposed system consists of m Primary NameNodes and many Backup

NameNodes to guarantee the fault-tolerance. It partitions the namespace into m fragments and manages them in a distributed manner using m Primary NameNodes. The number of Backup NameNodes depends on k, the redundancy factor of our architecture. We resolved the single point of failure (SPOF) problem, the most important issue in designing highly available Hadoop distributed file system, and the namespace limitation problem. In our system, the maximum size of the namespace can be easily extended by installing additional NameNodes. Moreover, our proposed architecture ensured the effective load balancing. We proved the consistency of the namespace regardless some failures which are occurred in our proposed architecture. We also evaluated our system's performance, and showed that the synchronization overhead, which is required for fault-tolerance, is small enough.

4.2 Future Directions

Regarding the proposed methods for the fault-tolerance, there exist some issues for improving our methods from both practical and theoretical viewpoints.

A Snapshot Algorithm To realize the concurrent execution, our new partial snapshot algorithm combines multiple snapshot algorithms which are initiated concurrently. However this combining process required the communications of many messages. The combining process makes the system's performance high, nonetheless, the performance can be improved more if we can reduce these communication messages.

Our snapshot algorithm decides the Main Initiator for termination-detecting when the collision is occurred. The Main Initiator detect the termination of the entire (combined) snapshot algorithm, and its behavior never affects the timing of each local checkpoint. Therefore we expect that the termination detection process can be implemented with distributed manner. This means that each initiator only detects its local (subsystem) termination, and shares only the result (termination) among initiators. This method allows no combining process, thus many communication messages are not required.

A NameNode Cluster for HDFS Our system uses multiple NameNodes for fault-tolerance. This improves the system performance and scalability. However, to exert the best performance of our proposed system, the effective partitioning rule is reuiqred. In our experiments, we simply partitions the namespace based on lexicographical target path. This may cause the disproportionate process load among NameNodes. Another future work is a dynamic partitioning policy which can improve the system performance more, because it makes the load balancing dynamic.

Acknowledgements

I have been fortunate to receive assistance from many people. I would like to express my gratitude to Professor Toshimitsu Masuzawa for his guidance and encouragement. And I would like to thank Dr. Tadashi Araragi for his helpful comments. I have also received valuable advice from Professors of the Graduate School of Information Science and Technology, Osaka University. Among them, I would like to extend my gratitude to Professor Kenichi Hagihara, Professor Shinji Kusumoto, and Associate Professor Hirotsugu Kakugawa for their precious comments and advice on this dissertation. I am also deeply grateful to Project Assistant Professor Junya Nakamura at Toyohashi University of Technology for his helpful advice. I would like to acknowledge Professor Katsuro Inoue and Professor Yasushi Yagi for their helpful comments on my work.

I wish to express my gratitude to Professor Masafumi Yamashita at Kyushu University, Professor Koichi Wada at Hosei University, Professor Yoshiaki Katayama at Nagoya Institute of Technology, Associate Professor Sayaka Kamei at Hiroshima University, Associate Professor Taisuke Izumi at Nagoya Institute of Technology, Assistant Professor Tomoko Izumi at Ritsumeikan University, Associate Professor Hiroyuki Nagataki at Okayama University, and Assistant Professor Yukiko Yamauchi at Kyushu University for their useful comments. In particular, I thank to Assistant Professor Fukuhito Ooshita for his time and kindness.

I wish to thank to all the members of Algorithm Engineering Laboratory, Graduate School of Information Science and Technology, Osaka University. Notably, I thank to Fusami Nishioka and Hisako Suzuki for their kind supports. I also appreciate Rotary International District 2660, Hattori International Scholarship Foundation, and Japan Student Service Organization (JASSO) for their economic support for my education in Japan.

I could not finish this dissertation if I were not encouraged by my best friends. It is so precious experience for me to meet with my friends. Especially, I deeply thank to Hyungsun Yoon, Hyungshin Choi, Sanghoon Kim, Woong Kang, and Seungyong Bong for their kind supports. I am always encouraged by Yongseok Chae, Seonghyun Kim, and Chanshik Lim, who were the

members of Samsung Software Membership. I would like to feel gratitude to Eunjong Choi for her kindness. I am also deeply grateful to Mina Hirano and Yusuke Maenaka for their hospitalities and many pieces of advice in my life in Japan.

Finally, I deeply appreciate my parents, Heechul Kim and Soonok Kim, and all of my family for their supports and kindness during my life.

Bibliography

[1] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms and Systems*. Cambridge University Press, 2011.

[2] A. D. Kshemkalyani et al. Fast and message-efficient global snapshot algorithms for large-scale distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(9):1281–1289, 2010.

[3] A. Mostefaoui et al. From static distributed systems to dynamic systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 109–118, 2005.

[4] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. the second edition, Pearson, 2007.

[5] Andrew Ryan, Facebook. Under the hood: Hadoop distributed filesystem reliability with namenode and avatarnode, 2012.

[6] B. Randell. System structure for software fault-tolerant. *IEEE Transactions on Software Engineering*, 1:220–232, 1975.

[7] Barry W. Johnson. *An Introduction to the Design and Analysis of Fault-Tolerant Systems*. Prentice Hall, 1995.

[8] Big data. `http://en.wikipedia.org/wiki/Big_data`.

[9] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the IEEE International Symposiuim on Reliability Distributed Software and Database*, pages 207–215, 1984.

[10] D. F. Carcia et al. Benchmarking of web services platforms - an evaluation with the tpc-app benchmark. In *Proceedings of the International Conference on Web Information Systems ans Technologies (WEBIST)*, pages 11–13, 2006.

[11] D. Molkov. Hot standby for namenode. Technical report, Apache Hadoop HDFS Issues, HDFS-976, 2013.

[12] Dean J. et al. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[13] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26, 2008.

[14] F. Mattern et al. Efficient algorithm for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4), 1993.

[15] Ghemawat S. et al. The google file system. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[16] S. Ghosh. *Distributed systems: an algorithmic approach.* Chapman & Hall/CRC, 2007.

[17] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* 2nd Edition, Wiley, 2004.

[18] J. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east, 2012. IDC iView.

[19] J. M. Helary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems*, pages 183–190, 1997.

[20] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[21] K. Shvachko. Giraffa file system, a distirbuted highly available file system. `https://code.google.com/a/apache-extras.org/p/giraffa/wiki/Introduction`.

[22] K. Shvachko. Name-node memory size estimates and optimization proposal. Technical report, Apache Hadoop Common Issues, HADOOP-1687, 2007.

[23] K. Shvachko. Hdfs scalability: the limits to growth. *;login:*, 35(2):6–16, 2010.

[24] K. Shvachko. Scalability of the hadoop distributed file system. Technical report, Hadoop Blog in Yahoo! Developer Network, 2010.

[25] K. Shvachko. Warm ha namenode going hot. Technical report, Apache Hadoop Issues, HDFS-2064, 2011.

[26] K. Shvachko et al. The hadoop distributed file system. In *Proceedings of IEEE the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[27] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[28] M. Elnozahy et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing surveys*, 34:375–408, 2002.

[29] M. J. Fischer, N. D. Griffeth, and N. A. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3), 1982.

[30] M. P. Herlihy et al. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, 1990.

[31] M. Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013.

[32] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[33] R. Baldoni, F. Quaglia, and B. Ciciani. A vp-accordant checkpointing protocol preventing useless checkpoints. In *Proceedings of International Symposium on Reliable Distributed Systems*, pages 61–67, 1998.

[34] R. Garg et al. Efficient algorithms for global snapshots in large distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):620–630, 2010.

[35] R. Garg, V. Garg, and Y. Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 269–277, 2006.

[36] R. Garg, V. Garg, and Y. Sabharwal. Efficient algorithms for global snapshots in large distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):620–630, 2010.

[37] R. Koo and S. Toueg. Checkpointing and roll-back recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[38] R. Prakash and M. Singhal. Maximal global snapshot with concurrent initiators. In *Proceedings of the 6th IEEE Symposium of Parallel and Distributed Processing*, pages 334–351, 1994.

[39] Replication (computing). `http://en.wikipedia.org/wiki/Replication_%28computing%29`.

[40] RosettaNet. `http://www.rosettanet.org`.

[41] S. Moriya and T. Araragi. Dynamic snapshot algorithm and partial rollback algoithm for internet agents. In *Proceedings of the 15th International Symposium on DIStributed Compuiting(DISC 2001) Brief Announcecments*, pages 23–28, 2001.

[42] S. Moriya and T. Araragi. Dynamic snapshot algorithm and partial rollback algoithm for internet agent (in japanese). *IEICE Transactions on Information and Systems*, J86-D-I:301–317, 2003.

[43] T. H. Lai and T. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[44] T. White. *Hadoop: The Definitive Guide*. 3rd edition, O'Reilly Media, Yahoo! Press, 2012.

[45] The Apache Software Foundation. Apache hadoop. `http://hadoop.apache.org/`.

[46] The Apache Software Foundation. Apache hbase. `http://hbase.apache.org/`.

[47] The Apache Software Foundation. Apache zookeeper. `http://zookeeper.apache.org/`.

[48] The Apache Software Foundation. Hdfs high availability. `http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithNFS.html`.

[49] Todd Lipcon. Quorum-based protocol for reading and writing edit logs. Technical report, Hadoop HDFS Issues, HDFS-3077, 2012.

[50] T.T.-Y. Juang and S. Veckatesan. Crash recovery with little overhead. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 454–461, 1991.

[51] W3C. World wide web consortium. `http://www.w3.org/`.

[52] W3C Working Group Note. Web services architecture, 2004. `http://www.w3.org/TR/ws-arch/`.

[53] W3C Working Group Note 11. Web services architecture requirements, 2004. `http://www.w3.org/TR/wsa-reqs/#id2604831`.

[54] Y. Kim et al. Brief announcement: A concurrent partial snapshot algorithm for large-scale and dynamic distributed systems. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security (SSS 2011)*, pages 445–446, 2011.