Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 2, Number 1, pages 4–11, January 2013

Evolving fault-tolerance in Hadoop with robust auto-recovering JobTracker

Nobuyuki Kuromatsu
The Graduate School of
Information Science and Technology of
Osaka University
Osaka, Japan 565–0871
Email: n-kuromt@ist.osaka-u.ac.jp

Masao Okita
The Graduate School of
Information Science and Technology of
Osaka University
Osaka, Japan 565–0871
Email: n-kuromt@ist.osaka-u.ac.jp

Kenichi Hagihara
The Graduate School of
Information Science and Technology of
Osaka University
Osaka, Japan 565–0871
Email: hagihara@ist.osaka-u.ac.jp

Abstract—**Hadoop is a popular open source software for supporting a large scale distributed data processing. While it achieves high reliability, the job scheduler, named JobTracker, remains the single point of failure. If the JobTracker fails to stop during a job execution, the job is canceled immediately and all of intermediate results are lost. We propose an auto-recovery system against the fail-stop without additional hardware. Our recovery mechanism is based on a checkpoint method. A snapshot of the JobTracker is stored on a distributed file system periodically. When the system detects the fail-stop by using timeout, it automatically recovers the JobTracker by a snapshot. The key feature of our system is a transparent recovery such that a job execution continues during a temporary fail-stop of the JobTracker and completes itself with a little rollback. The system achieves fault-tolerance for the JobTracker with overheads less than 4.3% of the total execution time. It reduces the reassigned tasks caused by a rollback compared to a naïve rollback.**

Index Terms—**Fault-torelance, checkpoint and recovery, Hadoop, master-worker**

I. Introduction

Recently, MapReduce programming model[1] has become a promising paradigm for a parallel and distributed large data processing. Hadoop[2] is a master-worker framework that supports an implementation of MapReduce on a cluster environment. Many companies like IBM, Amazon, Yahoo!, etc. apply Hadoop to *Big data*[3] analysis since it enable us to process huge data, tens of terabytes to petabytes, with low cost and high performance.

Hadoop has two master-worker systems. The first system is MapReduce framework (MRFW), that processes large data on many machines. The other is Hadoop distributed file system (HDFS), specialized in handling large data stream. Each system consists of a single master and multiple workers.

A key feature of Hadoop is high reliability. A parallel and distributed system requires fault-tolerance against machine failure[4]. Since processing huge data requires many machines and a long time execution, the probability that one of machines causes failure is relatively high. In order to achieve fault-tolerance, Hadoop multiplexes workers of both HDFS and MRFW. Besides, Hadoop stores an image file of HDFS periodically against failure on the master of HDFS.

However, JobTracker (JT), the master of MRFW, remains the single point of failure in Hadoop. Original developers assume that fault-tolerance for the JT is unnecessary because the specific single node in a machine cluster, namely JT, rarely fails to stop in general. If a job unexpectedly terminates, a user requires restarting the job from the beginning. Therefore, a failure after processing almost of an execution causes enormous unnecessary time to obtain the result. It makes it difficult to apply Hadoop to time-critical applications, such as core banking solutions.

NTT DATA reported a document[5] about fault-tolerance for the JT. They apply Kemari software FT[6] to Hadoop. They execute Hadoop on a cluster of virtual machines (VM). To achieve fault-tolerance for JT, the VM which invokes the JT process synchronizes its memory with a spare VM. When the machine that the primary VM runs on fails to stop, Kemari migrates the JT process to the spare VM instantly. Hadoop continues to execute a job without detecting the failure. However, Kemari requires additional machine and network between the primary VM and the spare VM for memory synchronization. Moreover, Kemari does not tolerate for software failure because memory contents of the primary VM and the spare VM are exactly the same. The system tolerates for only single failure since the number of spare VM is one for each VM.

In this paper, we propose an auto-recovery system for the JT in order to apply Hadoop to applications that expect a job completion in fixed time. To achieve user-transparent fault-tolerance for JT, we implement two techniques based on Hadoop 0.20.2. The first is checkpoint and recovery for the JT. Our system also reconstructs the new JT according to states of TaskTrackers to avoid rolling them back. Secondly, we use timeout mechanism in order to detect failure on the JT automatically. Note that users require no modification of their programs to apply our system.

Comparing to the system proposed by NTT DATA, our auto-recovery system requires no additional hardware. Our system tolerates for not only hardware failure but also software failure. It also allows repeatedly fail-stop of the JT. However, it causes the overheads of checkpoints duration execution and the overheads of recovering the JT at a failure.

The rest of this paper is organized as follows. Section II introduces the overview of Hadoop. Section III and Section IV

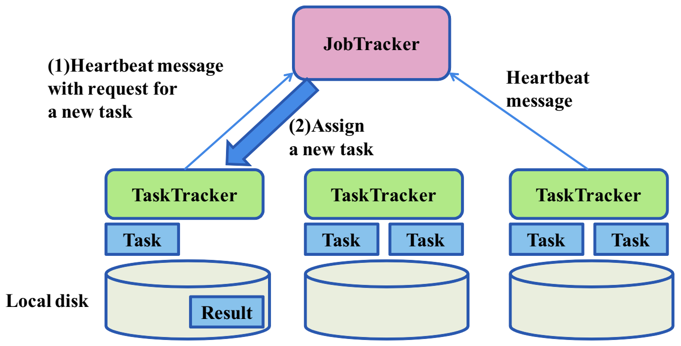

Fig. 1. An overview of task processing between the JobTracker and TaskTrackers. The size of task slot is two.

describes our goal and our auto-recovery system, respectively. In Section V, we show the experimental results. Section VI concludes the paper.

II. Hadoop

Here, Namenode (NN) and Datanode (DN) are a master process and a worker process of HDFS, respectively. TaskTracker (TT) is a worker process of MRFW. A pair of a DN and a TT runs on each machine node in a cluster and works collaboratively. On the other hand, there is the single pair of the NN and the JT in the cluster.

A. HDFS

HDFS scales its performance of loading and storing data depending on the number of DNs. HDFS divides data which a user intend to write into multiple certain sizes of data, called split. The NN assigns each split to a DN and manages the relation between the split and the DN. A DN stores assigned splits to local disk of the node on that the DN is running. When a user intends to read data, the NN gathers the splits which compose the data from DNs.

HDFS has fault-tolerance for both a DN and the NN. HDFS replicates splits to multiple DNs and stores a file image of the NN against a failure[7]. Since HDFS provides atomic operations, no data corruption occurs during writing.

B. MapReduce

MRFW divides a job into multiple tasks to execute them on distributed nodes in parallel. The JT schedules tasks and assigns them to TTs in consideration of data locality. A user sets the size of task slot, namely the number of tasks executing at once on each TT, through a configuration file of Hadoop. A TT sends a message, called heartbeat, to the JT periodically during a job execution to inform the progress of the tasks. If a TT completes a task, the TT sends a request for a new task within a heartbeat (Figure 1 (1)). When the JT receives a heartbeat including the request, it assigns a new task to the TT by replying of the heartbeat (Figure 1 (2)).

MapReduce tasks are classified into two types, MapTask and ReduceTask. A MapTask processes a split on HDFS and stores the result to local disk. A ReduceTask gathers a part of the result from each MapTask and summarizes them. A ReduceTask stores the result to HDFS. MRFW allows the JT to assign a redundant task to multiple TTs as a backup task. Backup tasks run on different nodes concurrently. The JT accepts the result of the first task to complete so that it could avoid waiting for the task completion which a low performance node executes.

MRFW has fault-tolerance for a TT. If a TT sends no heartbeat to the JT for a certain period, JT decides that the TT fails to stop. Then the JT reassigns the tasks that the failed TT has executed to other TTs.

If the JT fails to stop, MRFW terminates a job execution immediately. As long as the input data for the job is available on HDFS, a user can restart the job by rebooting JT process. Although partial results of tasks in the last execution remain on HDFS or local disk on each node, the JT ignores them and assigns the all tasks to TTs again.

III. Our Goal

In this study, we intend to complete job executions in fixed time even if the JT fails to stop. We provide completely automatic recovery from a failure so that users should not have to worry about whether the failure occurs. We also design the recovery system to minimize increase of the execution time of a job.

We assume a failure on the JT as a situation that the JT cannot reply to a heartbeat from TTs. The failure includes an unexpected termination of the JT process, an accidental shutdown of the node that the JT is running on, and a lost connection to the JT. We also assume that a failure probably repeats during a job execution but multiple failures do not occur at once.

We consider that HDFS has enough fault-tolerance. The NN and the JT generally run on the same node, but we execute the NN on a different node to avoid that a failure on the JT also terminates the NN. In this paper, we assume no failure occurs on the node that the NN runs on.

IV. The Proposed System

We consider the following two popular methods to achieve fault-tolerance for JT.

The first method is duplication. It multiplies the JT on distributed nodes. If the primary JT fails to stop, it is automatically or manually switched to a redundant JT. To keep consistency among multiple JTs, the method requires periodically synchronization during job execution. An advantage of this method is instant recovery from a failure. By contrast, a disadvantage is that it wastes computational resources and hardware for redundant JTs even if no failure occurs. Tolerance for multiple failures requires more wasted resource and hardware.

The second method is checkpoint and recovery. It periodically creates a snapshot of the current JT state and stores it to a storage. If the JT fails to stop, a new JT starts up and restores the state before the failure according to a snapshot. Although this method requires larger time for the recovery than the first method, it requires less additional resources and hardware.

We decide to use a checkpoint method because we focus on saving resources rather than instant recovery. A recovery time increases little execution time because all TTs continue operations during a failure and recovery. A checkpoint tolerates a failure of multiple times without additional hardware.

To construct an auto-recovery system for the JT, we implement three mechanism based on Hadoop: checkpoint on the JT, automatic detection of failure on the JT and recovery of the JT from failure.

A. *An issue in checkpoint and recovery*

Checkpoint and recovery approach usually requires a rollback. A rollback is an operation that returns the whole of system to the normal state at the last checkpoint. It abandons the operations that are executed during a period from the last checkpoint to a failure. The system must execute the abandoned operations again.

A rollback increases the execution time. In our proposed system, the JT requires rolling back for restoring the clean state. A naïve approach also rolls all TTs back to simplify keeping of the consistency between JT and them. Re-execution of the tasks abandoned by the rollback increases the execution time.

We focus on that TTs continue processing normally even if the JT temporary fails to stop. TTs should not be abandoned because they seem to be in normal state rather than a JT in failure. Therefore, we reconstruct the state of a new JT instead of rolling all TTs back.

B. *Checkpoint on the JT*

To reduce the size of a snapshot, we make a snapshot include minimum information necessary. Task scheduling on the JT requires the status of tasks (unassigned, processing, and completed), the status of TTs (busy or idle), the assignment table of tasks, and data allocation on HDFS. The JT should own the status and the assignment table because the NN provides information for data allocation. Therefore, a snapshot consists of only the status and the assignment table (see Appendix A for details). The JT owns other information, but they can be reconstructed from the environment and static configurations.

We implement event-driven checkpoint scheme rather than periodic checkpoint scheme. To decrease the overheads of checkpoints, we intend to decrease the frequency of checkpoints. We let the JT generate a snapshot whenever N tasks completed. A user can set N through a configuration file of Hadoop.

We also decide to store snapshots to HDFS because HDFS provides high availability and reliability. Data on HDFS is accessible from the all nodes, so that it allows all nodes to recover the JT with a snapshot. Besides, HDFS achieves fault-tolerance for the data by replication. A snapshot is available even if one replica is corrupt, one node causes failure or a part of network is unavailable.

Note that storing data to a storage requires serialization. Loading data from a storage also requires deserialization.

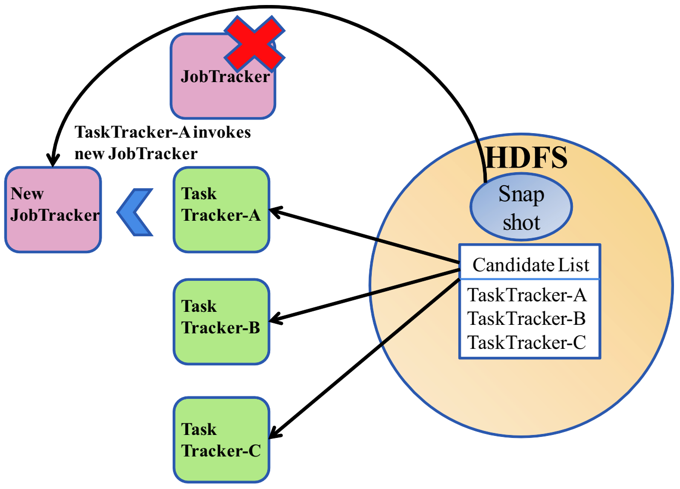

Fig. 2. An overview of switching JobTracker. After TaskTrackers detect a failure on JobTracker, they load the candidate list. The TaskTracker registered top of the list terminates itself and invokes a new JobTracker.

The serialization and deserialization of a snapshot increases the overheads of checkpoints and recovery. We discuss the overheads later in V-A.

C. *Automatic detection of failure*

We implement a timeout mechanism on a TT to detect failure on the JT automatically. Automatic detection generally uses heartbeat messages between two servers. We focus on a heartbeat between the JT and a TT originally implemented in Hadoop. If a TT fails to send a heartbeat to the JT for M times, the TT recognizes the JT fails to stop. After recovery of the new JT, TTs switch the destination of heartbeats to the new JT.

Note that a timeout mechanism may cause a misdetection due to temporary network congestion and other reasons. The misdetection divides a cluster into two node group: a group of TTs that detect a ghost failure and a group of TTs that detect no failure. The former group invokes a new JT to resolve the ghost failure. After that, the TTs in the former group send a heartbeat to the new JT. On the other hand, the original JT misunderstand these TTs in failure because it receives no heartbeat from them. As a result, the original JT assigns tasks to only the TTs in the latter group. Thus the misdetection degrades performance of job execution.

To avoid this performance degradation, we force the new JT to terminate the old JT. However, this method causes a redundant recovery every misdetection. We should detect a failure according to the majority in future work.

D. *Recovery from failure*

When our system detects the JT in failure, a specific TT terminates itself and invokes a new JT process. The new JT then loads the last snapshot from HDFS to recover the state of the old JT (Figure 2). Lack of one TT causes no serious problem for job executions due to the fault-tolerance for TT, but it decreases execution efficiency.

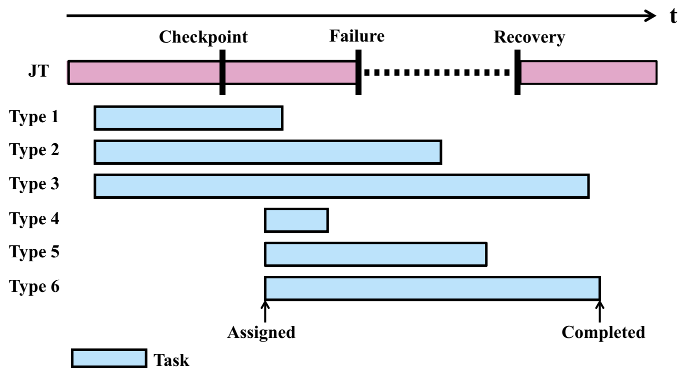

Fig. 3. The abandoned tasks by a rollback in naïve recovery approach. Timing of the assignment and the completion classify the tasks into six types.

Our system decides the TT that becomes a new JT according to a candidate list. The candidate list is a list of nodes on which a TT runs. The JT initializes the list at its start up and registers TTs to it in order of heartbeat arrival.

To be accessible from all TTs even with failure, the list is stored to HDFS as same as a snapshot.

Each TT detects a failure and then loads the list from HDFS independently. Here we call the top of the list the candidate node. The TT that runs on the candidate node becomes the new JT. If the candidate node is also down, the second of the list becomes next candidate node. The other TTs wait for a second and then tries to switch the destination of a heartbeat to the candidate node repeatedly. If TTs fails to switch for certain times, they try to switch the destination to the second candidate node. Thus, all TTs find the new JT without synchronization. This is an advantage of our system in terms of scalability.

Note that our system does not affect HDFS because the DN on the candidate node remains after recovery.

1) Continuing job execution: The original MRFW intends to cancel a job execution when it faces failure on the JT although the other processes is still running. The cancelation is due to JobClient, the gateway between users and MRFW. A user submits a job and confirms the progress of it through JobClient. JobClient requires a report of the progress from the JT periodically during job execution. If JobClient fails to receive the report, it immediately cancels to execute a job. It also cancel our recovery operation at that time.

To prevent this, we modify the implementation of JobClient. JobClient sends a dummy report to itself during recovery. After the candidate node invokes a new JT, the new JT sends a request for switching to JobClient. JobClient confirms the request by comparing to the candidate list and then switches a source of reports to the new JT.

2) Reconstructing the JT for transparent recovery: Since TTs continue operations during failure and recovery, information owned by the new JT based on the last snapshot probably conflicts that owned by TTs. As mentioned before, a naïve approach to avoid the conflict is rolling all TTs back. In contrast, our system directly resolves the conflict by reconstructing the new JT according to information from TTs to minimize the increase of the execution time.

Firstly, the new JT reconstructs the status of tasks before restarting the original operations. Target tasks are that TTs are executing in a period from the last checkpoint to the recovery of the new JT. Timing of the assignment and the completion classify the tasks into six types as shown in Figure 3. A task of type 3 causes no conflict because the JT accurately recognizes both the assignment and the completion. A task of type 2 also causes no conflict. The new JT recognizes the completion of the task even after recovery because the TT that completes the task sends a heartbeat to notice the completion repeatedly. In the case of the other types, a heartbeat from the TT includes a different task progress from that the new JT recognizes, resulting in conflict.

To resolve the conflict, we extend a TT to notify the current status of assigned tasks to the new JT just after recovery. The new JT changes its recognition according to the notices. If the notices include tasks of type 1 and type 4, the new JT recognizes they are completed. If they also include tasks of type 5 and type 6, the new JT recognizes they are in processing. As a result, type 5 and type 6 seem to be the same as type 2 and type 3, respectively. A task of type 5 and type 6 therefore no longer cause a conflict.

However, we have to re-execute the tasks executed on the candidate node. Since the TT on the candidate node has been terminated, it no longer completes tasks and notifies their completion. After reconstructing, the new JT reassigns the tasks to arbitrary TTs.

Secondly, TTs that execute ReduceTask also reconstruct their own information. Each ReduceTask has a list of completed MapTasks for gathering the result of completed tasks. To update the list, a ReduceTask periodically sends a request to the JT with the last index of the list. The JT replies to the request with a part of its own list after the index in the request. This algorithm assumes that the order of the list is the same between the JT and ReduceTasks. However, after rolling the JT back, this algorithm causes an error because the order is probably difference. To avoid the error, a ReduceTask synchronizes its own list to that of the JT when the TT sends the first heartbeat to the new JT.

V. Evaluation

This section shows experiments for evaluating overheads of our auto-recovery system. We compare the total execution time of our system to the original Hadoop both in the case without failure and with failure.

An experimental environment is a computer cluster that consists of 57 nodes interconnected with gigabit ethernet. Each node has an Intel Xeon 3.4 GHz dual core CPU, 2GB RAM, and 80GB disk. We base our system on Cent OS 5.0 and Hadoop 0.20.2.

A. The overheads of checkpoints

We measure the total execution time of a job, which executes the WordCount[8] sample program of Hadoop, under the situation that no failure occurs on the JT during the

TABLE I
THE TOTAL EXECUTION TIME OF A JOB ON OUR SYSTEM WITH VARYING EXPERIMENTAL PARAMETERS IN THE CASE WITHOUT FAILURE. EACH VALUE IS THE AVERAGE OF THREE MEASUREMENTS. P DENOTES THE NUMBER OF TTS.

	$P = 19$			$P = 38$			$P = 57$		
	w/o checkpoint	$N = 10$	$N = 100$	w/o cp.	$N = 10$	$N = 100$	w/o cp.	$N = 10$	$N = 100$
18GB	987	990	986	510	516	513	365	372	368
36GB	1,860	1,873	1,860	977	985	982	640	648	643
54GB	2,794	2,826	2803	1,386	1,414	1,385	892	920	902
72GB	3,670	3,745	3,681	1,830	1,882	1,827	1,172	1,223	1,174

TABLE II
THE TOTAL EXECUTION TIME OF A JOB AND DOWNTIME OF THE JOBTRACKER IN THE CASE WITH A FAILURE AND AUTO-RECOVERY. THE JOB EXECUTES A WORDCOUNT PROGRAM FOR 18GB INPUT DATA WITH $P = 19$, $N = 10$, AND $M = 3$. A FAILURE OCCURS IN A SPECIFIC PHASE AT ONCE. TIMES ARE PRESENTED IN SECOND.

	phase at the failure	progress at the failure: Map	progress at the failure: Reduce	elapsed time at the faiure	downtime of the JT	the total execution time	# of reassigned tasks: proposed	# of reassigned tasks: naïve
(a)	(no failure)	-	-	-	-	990	-	-
(b)	M	13%	0%	131	27	1,004	2	46
(c)		13%	0%	131	28	1,020	2	44
(d)		13%	0%	131	26	1,019	2	45
(e)	MR	80%	23%	734	27	1,080	12	39
(f)		80%	22%	738	28	1,080	12	41
(g)		80%	25%	779	27	1,112	12	41
(h)	R	100%	32%	975	27	1,019	16	22
(i)		100%	32%	963	29	1,038	16	9
(j)		100%	33%	981	28	1,025	16	20

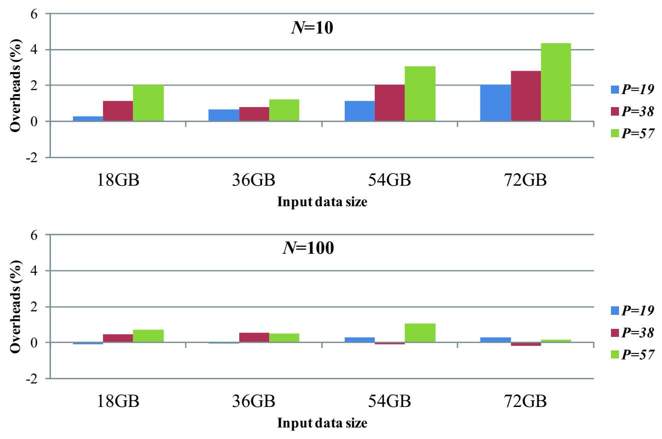

Fig. 4. Estimated overheads of checkpoints.

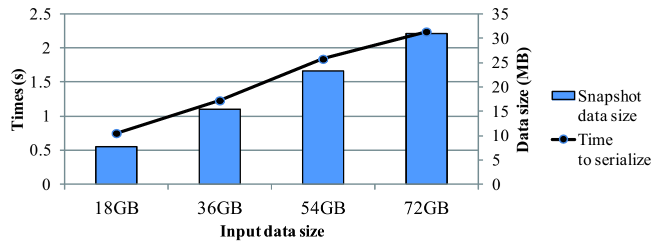

Fig. 5. The size of a snapshot and its serializing time varying the input data size ($P = 19$ and $N = 10$).

execution. We also evaluate the size of a snapshot generated by checkpoints.

Table I shows the execution time with varying experimental parameters: the frequency of checkpoints, the number of TTs, and Input data size. We can estimate the overheads by comparing the result of $N = 10$ and $N = 100$ with that without checkpoints because the case without checkpoints is similar to original Hadoop.

Estimated overheads of checkpoints are shown in Figure 4. The overheads are less than 4.3%. High frequency of checkpoints increases the overheads. In the cases with $N = 10$, the overheads also increases as P increases. On the other hand, we find no correlation between P and the overheads in case with $N = 100$. Our checkpoint method principally increases loads on the JT, while the JT usually idles in original MRFW. Therefore, the loads does not necessarily increase the execution time if the frequency of checkpoints is enough low.

The execution time slightly decreases rather than that without checkpoints in four cases. We consider this is due to originally perturbation in Hadoop. Regardless our auto-recovery methods, the execution time of a job frequently varies in ten seconds according to data allocation on HDFS. Since Hadoop automatically allocates data to HDFS, we cannot control it in this experiment.

Figure 5 indicates that the size of a snapshot is proportional to the input data size. Almost of a snapshot is information on the status of tasks which compose a job. Since MRFW divides the input data into certain size splits and assigns a task to each split, the number of tasks increases as input data size

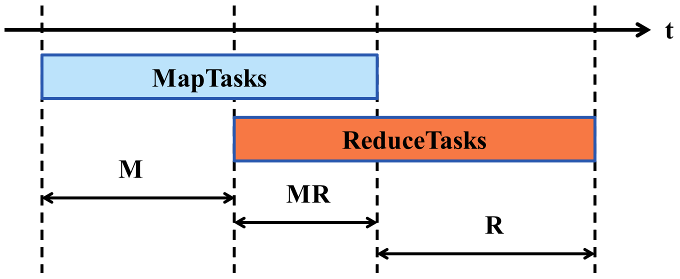

Fig. 6. Classified execution phases according to a kind of tasks in processing: phase M processing only MapTasks, phase MR processing both MapTasks and ReduceTasks, and phase R processing only ReduceTasks.

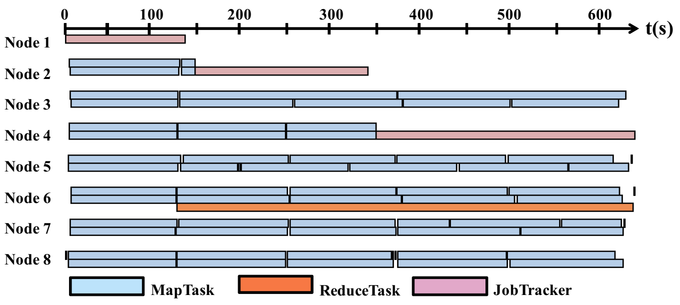

Fig. 7. An experimental result of job behavior with failures and recoveries. Two failures occur at the different time. Node 2 and node 4 invoke a new JobTracker in the middle of the execution to recovery from the failures.

increases. Therefore, the input data size determines the size of a snapshot.

Large snapshots probably increase the overheads of our system. To create a snapshot safely, we should lock the original process of the JT while the JT serializes a snapshot and stores it. This lock causes delay in replying to a heartbeat. A large snapshot makes the delay large. If a TT sends a heartbeat includes a request for task assignment during the lock, the TT must wait until it receives reply, resulting in overheads. As shown in Figure 4, large input data increases the overheads in the cases with $N = 10$.

B. A test of auto-recovery

As shown in Figure 6, we classify progress of a job execution into three phases according to a kind of tasks in processing. While the JT tracks both MapTasks and ReduceTasks at phase MR, it tracks only MapTasks and only ReduceTasks at phase M and R, respectively.

To confirm auto-recovery mechanism, we force a failure to occur on the JT by *kill* system call at each phases. For example, Figure 7 shows behavior of a job with two failures at phase MR. The job executes a WordCount program for 5GB input data with $P = 7$ and $N = 10$. We first cause a failure on node 1 and then cause another failure on node 2 after recovery from the first failure. As shown in figure 7, even if a JT fails to stop, a new JT starts up on another node within 20 seconds.

Thus, our system can recover from a failure of multiple times. At the other phases, our auto-recovery system works surely as well as at phase MR.

C. The overheads of recovery

We measure the execution time of a job in the case with a failure and auto-recovery (see Table II). As same as V-B, the failure be caused deliberately. We confirmed the result of a job is correct in the all cases. Table II shows the increased execution time is up to 112 seconds while the downtime is up to 29 seconds.

Recovering the JT degrades execution efficiency because the number of TTs decreases by one to invoke a new JT. For example, in the case of phase M in Table II, the rest of the execution time after the failure is expected as $990-131 = 859$ seconds. The execution efficiency decreases to approximately $18/19 = 94.7\%$. Therefore we estimate roughly the total execution time at $131 + (819/0.947) = 1,038$ second. If the number of TTs is large enough, we consider the degradation of execution efficiency is negligible.

Although an early failure increases the rest of the execution time, the total execution time in the case of phase MR and R is larger than that of phase M in Table II. This is due to the increase of reassigned tasks. We discuss the problem later in V-C2.

1) Discussion about downtime: We define downtime T_u as an absence period of a JT, from the occurrence of a failure on a JT to a new JT in operation. The downtime depends on the following three operations. Table III shows the breakdown of downtime measured with changing input data size for a WordCount program.

The first is a detection of the failure by using timeout. In this experiment, we set $M = 3$. An interval of heartbeats is less than three seconds in default, so that the detection time T_f is around eight seconds. Users can control T_f by changing M.

The second is invoking a new JT process. T_i in Table III shows duration from the detection of the failure by the first TT to the completion of start up on the new JT. In our experimental environment, it is less than five seconds.

The last is a deserialization of a snapshot. As mentioned in V-A, the size of a snapshot is proportional to the input data size. Since large snapshot increases deserializing time T_d, it depends on the input data size. Deserializing time is larger than serializing time for the same data because a deserialization requires the generation of objects.

Note that the downtime does not necessarily increase the total execution time since TTs continues to execute tasks even if the JT fails to stop. We find no correlation between the downtime and the increased execution time.

2) Discussion about reassigned tasks: As shown in Table II, except the case (i), our system reduces the number of reassigned tasks compared to a naïve rolling back. This indicates that reconstructing JT prevents to rollback TTs.

Although the ideal number of reassigned tasks is at most four in this experiment, our system executes more reassigned tasks in the cases from (e) to (j). This is due to an unsophisticated implementation. The expected implementation should reassign the tasks that are completed on the candidate node after the last checkpoint. In addition, the current

TABLE III
THE BREAKDOWN OF DOWNTIME OF THE JOBTRACKER IN SECONDS ($P = 19$, $N = 10$, AND $M = 3$). T_f, T_i, AND T_d DENOTE TIME FOR DETECTING FAILURE, TIME FOR INVOKING A NEW JOBTRACKER, AND TIME FOR DESERIALIZING A SNAPSHOT, RESPECTIVELY.

Input data	snapshot	downtime	breakdowns		
(GB)	(MB)	T_u	T_f	T_i	T_d
18	7.1	27.6	7.5	3.6	15.4
36	13.8	41.0	8.1	4.8	28.9
54	20.6	54.4	8.4	3.5	42.2

TABLE IV
THE NUMBER OF EXECUTED TASKS INCLUDING THE REASSIGNED TASKS. THE JOB EXECUTES A WORDCOUNT PROGRAM FOR 18GB INPUT DATA WITH $P = 19$, $N = 10$, AND $M = 3$. THE NUMBER ORIGINALLY VARIES DUE TO BACKUP TASKS.

	phase at the failure	# of executed tasks	# of reassigned tasks
(a)	(no failure)	305	-
(b)		301	2
(c)	M	298	2
(d)		300	2
(e)		315	12
(f)	MR	314	12
(g)		319	12
(h)		305	16
(i)	R	293	16
(j)		305	16

implementation also reassigns the all tasks that have been completed on the candidate node until a failure. Therefore, a late failure increases reassigned tasks (see Table IV). With expected implementation, the number of reassigned tasks in the case (i) would be smaller than a naïve rolling back.

In the cases from (e) to (g), the excess reassigned tasks increases the execution time. Table IV indicates that the number of executed tasks in these cases increases due to the reassigned tasks. On the other hand, in the case from (h) to (j), the execution time and the number of executed tasks increase a little in spite of more excess reassigned tasks. The all of the reassigned tasks are MapTasks. All TTs continue to execute ReduceTasks even with a failure, because they have already gathered the result of all MapTasks before the failure in these cases. If TTs complete all ReduceTasks before the completion of reassigned tasks, the reassigned tasks are canceled immediately. Therefore, the reassigned tasks slightly increase the execution time in these cases.

VI. CONCLUSION

We implements an auto-recovery system for the JobTracker, which is the single point of failure in Hadoop, to apply Hadoop to time-critical applications. The assumed failure is a situation that the JobTracker cannot reply to a heartbeat from TaskTrackers. Our system can recover the JobTracker from a failure of multiple times.

The system periodically stores a snapshot of the current JobTracker state to HDFS during a job execution. If once a failure occurs on the JobTracker, a specific TaskTracker switches new JobTracker. The new JobTracker loads a snapshot from HDFS and reconstructs the states of JobTracker according to the snapshot and information on the other TaskTrackers.

In order to detect a failure, each TaskTracker observes a timeout of a heartbeat that a TaskTracker send to the JobTracker periodically. This technique probably involves a misdetection due to a temporary network congestion. Therefore, the new JobTracker forces the old JobTracker to terminate immediately after its start up to prevent that two JobTrackers exist at the same time.

Experimental results show that the overheads of checkpoints are less than 4.3%. They also show that a job execution continues and finishes normally even with failures and recoveries. Our system reduces the reassigned tasks caused by a rollback compared to a naïve rollback. However, the recovered execution time increases due to an unsophisticated implementation.

ACKNOWLEDGMENT

REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[2] Leons Petrazickis and Bradley Steinfeld. Crunching big data in the cloud with hadoop and biginsights. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '11, pages 334–335, Riverton, USA, 2011. IBM Corp.

[3] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media/Yahoo Press, Sebastopol, CA, 2nd edition, 2009.

[4] Toshio Suganuma, Akira Koseki, Kazuaki Ishizaki, Yohei Ueda, Ken Mizuno, Daniel Silva, Hideaki Komatsu, and Toshio Nakatani. Distributed and fault-tolerant execution framework for transaction processing. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 2:1–2:12, New York, USA, 2011. ACM.

[5] NTT DATA CORPORATION. The accomplishment report of software engineering work cooperated among industry and academia in 2009. Technical report, Ministry of Economy, Trade and Industry, 2010. (In Japanese).

[6] Yoshi Tamura. Kemari: Virtual machine synchronization for fault tolerance using domt. In *Proceedings of Xen Summit Boston 2008*, Boston, USA, June 2008.

[7] Garhan Attebury, Andrew Baranovski, and Ken Bloom. Hadoop distributed file system for the grid. In *Proceedings of 2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*, NSS/MIC '09, pages 1056–1061, Orlando, USA, January 2009. IEEE.

[8] Wei Jiang, Vignesh Ravi, and Gagan Agrawal. Comparing map-reduce and freeride for data-intensive applications. In *Proceedings of 2009 IEEE International Conference on Cluster Computing and Workshops*, CLUSTER '09, pages 1–10, New Orleans, USA, September 2009.

APPENDIX A
DETAILS OF SNAPSHOT

A snapshot contains the following fields in *JobTracker* class:

- `int totalSubmissions`
- `int numResolved`
- `int totalMaps`
- `int totalReduces`
- `State state`
- `Map<JobID,JobInProgress> jobs`
- `TreeMap<String,ArrayList<JobInProgress>> userToJobsMap`
- `Map<String,Set<JobID>> trackerToJobsToCleanup`
- `Map<String,Set<TaskAttemptID>> trackerToTasksToCleanup`
- `Map<TaskAttemptID,TaskInProgress> taskidToTIPMap`
- `TreeMap<TaskAttemptID,String> taskidToTrackerMap`

- `TreeMap<String,Set<TaskAttemptID>> trackerToTaskMap`
- `TreeMap<String,Set<TaskAttemptID>> trackerToMarkedTasksMap`
- `Map<String,HeartbeatResponse> trackerToHeartbeatResponseMap`
- `Map<String,Node> hostnameToNodeMap`
- `Map<String,Integer> uniqueHostsMap`
- `JobTracker.RecoveryManager recoveryManager`
- `TreeSet<TaskTrackerStatus> trackerExpiryQueue`
- `FaultyTrackersInfo faultyTrackers`
- `String trackerIdentifier`
- `int totalMapTaskCapacity`
- `int totalReduceTaskCapacity`
- `int numTaskCacheLevels`
- `Set<Node> nodesAtMaxLevel`
- `TaskScheduler taskScheduler`
- `List<JobInProgressListener> jobInProgressListeners`
- `int numBlackListedTrackers`
- `HashMap<String,TaskTrackerStatus> taskTrackers`
- `JobConf conf`