Title: PAReS: A Proactive and Adaptive Redundant System for Enhancing MapReduce Job Completion Reliability and Service Quality

Authors: Jia-Chun Lin, Fang-Yie Leu, Ying-ping Chen

Affiliation:

Jia-Chun Lin & Ying-ping Chen	Department of Computer Science, National Chiao Tung University, Taiwan.
Fang-Yie Leu	Department of Computer Science, TungHai University, Taiwan.

Address:

Jia-Chun Lin	Reinhold-Frank-Str. 48c, 76133 Karlsruhe , Germany
Ying-ping Chen	Office EC 711, 1001 Ta Hsueh Road, HsinChu City 300, Taiwan
Fang-Yie Leu	Office ST422, No.1727, Sec.4, Taiwan Boulevard, Xitun District, Taichung 40704, Taiwan

Corresponding authors:

Ying-ping Chen, Tel: +886-3-5712121 ext 31446, Fax: +886-3-5724176, ypchen@cs.nctu.edu.tw

Fang-Yie Leu, Tel: +886-4-23590121 ext 33815, leufy@thu.edu.tw

Abstract: Recently, MapReduce has been a key and popular technology for tackling data-intensive applications. But its two master servers in current MapReduce implementations have a single-failure problem, which may interrupt MapReduce operations and filesystem services. To solve this problem, several redundant schemes have been proposed. However, some schemes cannot deliver good service quality, which in this study is defined as short service downtime and seamless/complete takeover. Some schemes have problems such as delayed synchronization, high energy consumption, and expensive synchronization cost, consequently failing to provide good service quality in a cost-efficient manner. Some others are unable to provide a sufficiently high reliability to complete a set of MapReduce jobs (we call it job completion reliability). Therefore, in this paper, we propose a hybrid takeover scheme, called the Proactive and Adaptive Redundant System (PAReS for short), which employs three service-quality improvement mechanisms, including a proactive synchronization and replication method, a mutual life monitoring algorithm, and an adaptive warm-up mechanism, to mitigate the above problems. The formal analysis and extensive experiments show that the PAReS improves job completion reliability and enhances service quality at acceptable energy consumption level and synchronization cost as compared with four state-of-the-art schemes.

Keywords: MapReduce, job completion reliability, redundant system, service downtime, takeover, proactive synchronization and replication, mutual life monitoring, adaptive warm-up mechanism

PAReS: A Proactive and Adaptive Redundant System for Enhancing MapReduce Job Completion Reliability and Service Quality

Jia-Chun Lin[1], Fang-Yie Leu[2], Ying-ping Chen[1]
[1]Department of Computer Science, National Chiao Tung University, Taiwan
[2]Department of Computer Science, TungHai University, Taiwan

Recently, MapReduce has been a key and popular technology for tackling data-intensive applications. But its two master servers in current MapReduce implementations have a single-failure problem, which may interrupt MapReduce operations and filesystem services. To solve this problem, several redundant schemes have been proposed. However, some schemes cannot deliver good service quality, which in this study is defined as short service downtime and seamless/complete takeover. Some schemes have problems such as delayed synchronization, high energy consumption, and expensive synchronization cost, consequently failing to provide good service quality in a cost-efficient manner. Some others are unable to provide a sufficiently high reliability to complete a set of MapReduce jobs (we call it job completion reliability). Therefore, in this paper, we propose a hybrid takeover scheme, called the Proactive and Adaptive Redundant System (PAReS for short), which employs three service-quality improvement mechanisms, including a proactive synchronization and replication method, a mutual life monitoring algorithm, and an adaptive warm-up mechanism, to mitigate the above problems. The formal analysis and extensive experiments show that the PAReS improves job completion reliability and enhances service quality at acceptable energy consumption level and synchronization cost as compared with four state-of-the-art schemes.

Keywords: MapReduce, job completion reliability, redundant system, service downtime, takeover, proactive synchronization and replication, mutual life monitoring, adaptive warm-up mechanism

1. INTRODUCTION

MapReduce [1] is a flexible distributed programming model introduced by Google to solve data-intensive applications. With this model, Google processes over 20 petabytes of data per day [1]. Apache, inspired by MapReduce, developed an open-source framework called Hadoop [2], which has been adopted by many companies and organizations, such as Facebook and Yahoo, to process their large-scale data. Other MapReduce implementations can be found in [3][4][5].

Generally, a MapReduce implementation, e.g., Apache Hadoop [2], is running on a MapRedcue cluster consisting of a set of common machines, rather than a set of high-end machines. In this cluster, two machines act as master servers. One is JobTracker [2], which coordinates the execution of MapReduce jobs running on a MapReduce cluster. The other is NameNode [2], which manages the distributed filesystem of the cluster. Due to running on a single and common machine, JobTracker and NameNode may fail because of various reasons, such as hardware failure, software malfunction, and bad configuration [6]. In fact, Yahoo has experienced three NameNode failures caused by hardware problems [6]. Even though JobTracker and NameNode are run on reliable hardware, they mail fail some day. Consequently, MapReduce operations and filesystem services will be interrupted, and all running jobs may not be able to proceed and complete.

To solve this problem, Hadoop [2] adopted a warm-standby node (called checkpoint node) to periodically back up the state of NameNode. Apparently, the corresponding energy consumption is low since the checkpoint node does not stay awake all the time. However, due to the fixed checkpoint interval (e.g., one hour), the recent backup files may be unable to reflect the latest NameNode state. Hence, when NameNode fails, the checkpoint node cannot provide fast and seamless takeover. To speed up the takeover process, several hot-standby-based schemes [6][29][9] have been proposed, which employ one or more hot-standby nodes to maintain the up-to-date state of the MapReduce master server, thereby causing more energy consumption and synchronization cost. These schemes also suffer from a delayed synchronization problem, i.e., the metadata synchronization is delayed until the master performs an update operation rather than receiving the corresponding request from a client. Consequently, when the master fails, those unfinished update operations and MapReduce jobs may be interrupted. Besides, a hot-standby node may crash before its master server does, consequently failing to provide a sufficiently high reliability to complete a set of MapReduce jobs (we call it job completion reliability). Other hot-standby-based and load-sharing schemes [8][9][10] have similar problems.

In this paper, we propose a hybrid takeover scheme called the Proactive and Adaptive Redundant System (PAReS for short) for a MapReduce master server based on our previous work [30]. The PAReS employs a hot-standby node (HNode for short) and a warm-standby node (WNode for short) to preserve the advantages of hot-standby schemes (i.e., fast takeover) and the advantages of warm-standby schemes (i.e., low energy consumption and good job completion reliability). However, naively utilizing the HNode and WNode cannot solve the delay synchronization problem and improve service qualities, including reducing the impact of the master-server failure on client requests and shortening service downtime, which is defined as a time period from the moment when the master server fails to the moment when a standby node takes over for the master and is ready to process client requests.

Hence, the PAReS further employs three service-quality improvement mechanism to solve the above-mentioned problems and guarantee good service qualities. The first mechanism is a proactive synchronization and replication method, which starts replicating the master's metadata to the HNode whenever the master updates its metadata receives a write request from a client, rather than when performing or finishing the request. Thus, when the master fails, the HNode can provide a rapid takeover and reduce the failure impact on client requests. The second is a mutual life monitoring algorithm which enables the master and HNode to promptly detect each other's failure, adjust the status of the WNode, and initiate the corresponding takeover process to accelerate takeover process and shorten takeover time. The third is an adaptive warm-up mechanism, with which the WNode warms itself up autonomously every fixed time interval and dynamically when receiving a warm-up request from the master or HNode. Consequently, the WNode can adapt itself to the current system status and improve its takeover performance.

The key contributions of this study are as follows:

- The PAReS provides the MapReduce master server with a simple, reliable, and energy-efficient working environment. By employing the HNode, WNode, and three service-quality improvement mechanisms, the PAReS not only preserves the advantages of hot-standby-only schemes and warm-standby-only schemes, but also solves the shortcomings of these schemes.
- We achieve a comprehensive evaluation by comparing the PAReS with several state-of-the-art schemes, including the No-Redundant scheme (NR for short), which is the approach that Hadoop offers for its JobTracker [2], Hot-Standby-Only scheme (HSO for short) proposed by Wang et al.'s [9], and the warm-standby-only scheme (i.e., the checkpoint node) used by Hadoop [7]. In addition, we also consider the warm-standby-only scheme in two versions. One is with automatic takeover, i.e., a monitor node is employed to detect the master's failure and ask the warm-standby node to take over for the master (we call it an A-WSO scheme). The other version is without automatic takeover, implying no monitor node is utilized (we call it a WSO scheme). A fair comparison is achieved by comparing these schemes in terms of their job completion reliabilities, energy consumptions, synchronization costs, service downtimes, and impacts of the master-server failure on client-submitted requests.
- The formal reliability analysis shows that the PAReS dramatically improves the job completion reliability of the master server, and it provides higher job completion reliability than most of the other schemes. The energy consumption analysis also demonstrates that the PAReS effectively reduce the high energy consumed by the HSO and A-WSO, but the energy consumption of the PAReS is unavoidably higher than those of the NR and WSO. The experimental results demonstrate that the PAReS indeed provides better service quality at acceptable synchronization cost as compared with the other schemes.

The rest of this paper is organized as follows. Section 2 introduces the background and related work of this study. Section 3 details the PAReS and the three service-quality improvement mechanisms. The job completion reliability and energy consumption analyses are shown in Sections 4 and 5, respectively. The experimental results are presented and discussed in Section 6. Section 7 concludes this paper and outlines our future studies.

2. BACKGROUND AND RELATED WORK

This section describes the background and related work of this study.

2.1 Background

MapReduce allows users to specify a job *J* by using two functions: map and reduce. The map accepts a set of key-value pairs in one domain and generates a list of intermediate key-value pairs in another. The reduce function merges all intermediate key-value pairs of the same key to generate final result. Fig. 1 illustrates the execution flow of *J* on a MapReduce cluster consisting of JobTracker, NameNode, and multiple worker nodes. A client requests an ID for *J* through steps 1 and 2. After receiving worker locations from NameNode in steps 3 and 4, the input file of *J* is divided into fixed-size data blocks and stored on worker nodes in step 5. In steps 6 and 7, *J* is submitted to JobTracker, and JobTracker initiates *J*, respectively. After requesting and retrieving the data-block information of *J* from NameNode in steps 8 and 9, JobTracker in step 10 assigns each map (reduce) task of *J*'s to an available worker, named mapper (reducer). Before running its assigned task, a mapper/reducer needs to retrieve its required input by consulting NameNode. When finishing its task, a mapper stores the generated results on its disk and replies to JobTracker with the disk location. When all mappers complete their tasks, the reducers can start their assigned tasks. After all reducers finish their tasks, JobTracker informs the client of the completion of *J*.

Generally, if JobTracker fails, *J* cannot be submitted, assigned, performed, or completed. Also, a failed NameNode cannot help JobTracker to assign tasks, assist workers to obtain required resources, and so on. As a result, JobTracker and NameNode must work properly during the execution of *J*.

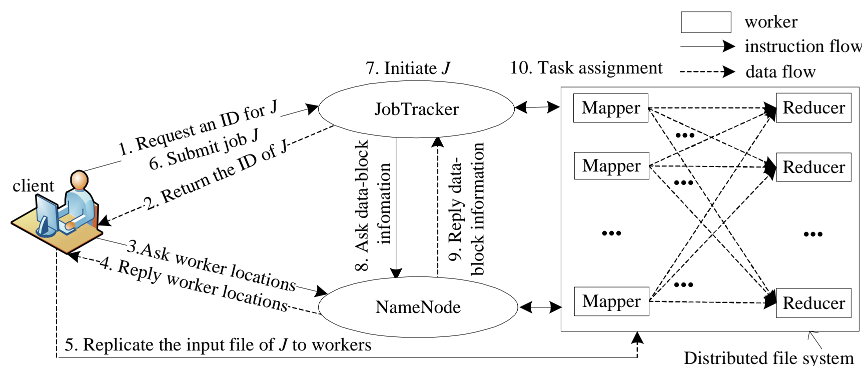


Fig 1. The execution flow of a MapReduce job *J* on a MapReduce cluster.

2.2 Related work

Redundant mechanisms, the common methods that a general system often uses to improve its system reliability, can be classified into four types: cold standby, warm standby, hot standby, and active parallel [12][13][14]. Current MapReduce implementation, e.g., Hadoop, only adopted a cold-standby mechanism to provide fault tolerance for its workers, rather than for its master server. This is because a cold-standby node does not maintain master-server states, and hence it cannot recover the master server to the latest state.

Some systems employed a warm-standby node as a backup server, to which the master server replicates its states periodically or under request. After that, the warm-standby node sleeps to reduce its failure probability and energy consumption. The checkpoint node [7] utilized by Hadoop is an example. The checkpoint node periodically generates a file to back up the log records of NameNode. When NameNode fails, the most recent file is used to restart it. In Hadoop, this action is performed manually by system managers, which might prolong takeover time and service downtime. To achieve an automatic takeover, a monitor node is required to detect the master's failure and request the checkpoint node to take over for the master when the master fails. But it increases the energy consumption and decreases the reliability improvement because the monitor node needs to stays awake all the time with the master. No matter the checkpoint node has the monitor node or not, due to the fixed checkpoint interval, the backup file may be out-of-date, implying that when NameNode crashes, the metadata produced after the generation of the file cannot be recovered, resulting in an incomplete takeover. Besides, in Hadoop, when the checkpoint node takes over for NameNode, it has to execute each of the operations recorded in its file and retrieve block-location metadata from all other workers. This often lengthens the takeover process and causes longer

service downtime. Other warm-standby-based schemes have similar problems, consequently failing to provide a fast and seamless takeover.

Several systems [2][7][8][9][15][16][29] utilized hot-standby mechanisms to speed up their failure recovery. Among these, the schemes proposed by [7][9][29] are designed for the MapReduce master server. The Hadoop backup node [7] synchronizes itself with NameNode. When NameNode fails, the backup node can quickly take over for it. But this node has three drawbacks. First, as mentioned previously, it may crash before NameNode does. Thus, the reliability improvement is limited. Second, the backup node cannot provide a short service downtime since it does not maintain block-location information (we call it transient metadata). When the backup node takes over for the master, it needs to enter a safe mode to retrieve all the transient metadata from workers, thereby prolonging the time to process user requests. Third, this node has the delayed synchronization problem, i.e., the synchronization occurs only when the backup node receives a journal stream, which is generated and sent by NameNode after NameNode finishes the corresponding operation. Hence, when NameNode crashes, the backup node cannot continue those unfinished operations, consequently affecting the corresponding job execution.

Wang et al. [9] proposed a metadata replication scheme to replicate metadata from a Hadoop master server to multiple hot-standby nodes. The authors also presented three configurable synchronization modes to make their scheme adapt to different workloads and network environments. When the master server fails, one of the hot-standby nodes will be elected to be the master server. Obviously, this scheme causes high energy consumption and expensive synchronization costs due to employing multiple hot-standby nodes. Besides, this scheme the same drawbacks incurred by the backup node [7]. Wan et al. [29] presented a hot-standby node to back up the state of JobTracker by synchronizing their job execution logs all the time, but this scheme also has the same abovementioned drawbacks.

Active parallel redundancy can be further classified into two types: Modular redundancy [17] and load-sharing redundancy [18]. The former, employing multiple redundant components/nodes to simultaneously perform a task, has been widely utilized in mission critical systems [19][20][21], but seldom employed in MapReduce. The latter, in which several nodes share the workload of a system, has been utilized at least by [22][23][24][10]. In this mechanism, a server informs other servers after updating its metadata so that its failure will not interrupt the operation of the entire system. Similar to all hot-standby-only schemes, this mechanism has limited reliability improvement and high energy consumption. Besides, its synchronization cost is even higher, which deviates the purpose of our PAReS. In our previous work [30], we proposed a hybrid redundant system call ReHRS to enhance the reliability and availability of the MapReduce master server. However, the synchronization method is complicated, and the failure detection is simplified such that the WNode cannot dynamically proceed its warm-up procedure to adapt current system state. Besides, the job completion reliability, energy consumption, and synchronization cost are not considered and analyzed.

3. THE PROPOSED SCHEME

In this section, we describe two types of state-update operations executed by MapReduce master servers and then introduce the three service-quality improvement mechanisms. Finally, the processes of taking over for the master server and HNode are presented.

3.1 State-update operations

In MapReduce, all state update-operations performed by a MapReduce master server (no matter it is NameNode or JobTracker), can be classified into two types: persistent and transient. A persistent update operation is a n-phase operation during which the master generates persistent metadata (P-meta for short), which is the metadata infrequently or never changed after it is generated, to update its state and log file, $n \geq 1$. For example, file access-right modification [6] is a one-phase operation, and job execution is a multi-phase one.

A transient update operation is an operation that the master needs to update its state, rather than needing to update its log file. When receiving transient metadata, which is the one frequently updated such as job execution state and data-block location sent by workers [6], the master server only update its in-memory state.

3.2 Proactive synchronization and replication method

Based on the primary/backup replication method [25][26] and chain replication technique [27], the proactive synchronization and replication method is designed to prevent any persistent update operation from being interrupted due to the occurrence of master-server failures, and enable the HNode to maintain all transient metadata of the master server so as to shorten service downtime. The basic idea is to replicate the master's metadata to the HNode whenever the master updates its P-meta and transient metadata.

Upon receiving a persistent update operation X from a client U, the master generates an prior log record (denoted by P_X), which consists of an incremental record number, X's ID (denoted by ID_x), and other necessary information about X. When the master initiates the first phase of X, it generates a initial log record (denoted by I_X) which comprises another incremental record number, ID_x, and information about the initialization of X. The synchronization processes for a prior log record and an initial log record are the same (see Fig. 2a), which work as follows:

Step 1) The master server inserts the log record into its log file.
Step 2) The master sends this record to the HNode and continues X.
Step 3) On receiving the record, the HNode inserts it into its log file.
Step 4) The HNode tells the master that it has received the log record.
Step 5) If the master does not receive the message after a predefined time period, it periodically sends the log record to the HNode until receiving the message or finding that the HNode has failed. The detailed failure detection will be described later.

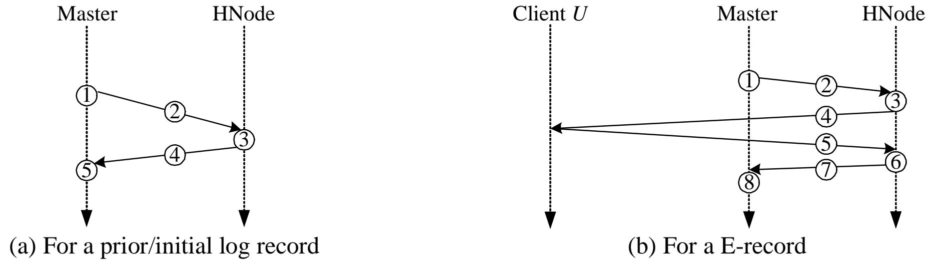


Fig. 2. The synchronization sequence charts when the persistent update operation X is issued by client U.

On the other hand, when the master finishes the G-th phase of X where $G = 1,2, \ldots, n$, it generates an end log record (E-record for short), denoted by E_X^G, which conveys a record number, ID_x, the G value, and the result of X's G-th phase (i.e., P-meta). The E-record synchronization process issued by user U as shown in Fig. 2b is as follows.

Step 1) The master inserts E_X^G into its log file and updates its state with the P-meta recorded in E_X^G.
Step 2) The master sends E_X^G to the HNode.
Step 3) On receiving E_X^G, the HNode inserts it into its log file and also updates its state. Now, the master and the HNode are synchronous. Note that the HNode updates its state with the received E-records one by one based on the increasing order of record numbers.
Step 4) The HNode delivers the P-meta to U.
Step 5) Upon receiving the P-meta, U responds the HNode.
Step 6) With the response from U, the HNode records {ID_x, G, "completed"} for reminding itself that someday when taking over for the master, it can ignore X's G-th phase.
Step 7) The HNode returns {ID_x, G, "completed"} to the master to show the completion of X's G-th phase.
Step 8) Upon receiving the message, the master also records this message. Otherwise, the master periodically sends E_X^G to the HNode until receiving the message or finding that the HNode has failed.

Notice that if X is issued by the master, the E-record synchronization process comprises only the steps 1, 2, 3, 7, and 8 since no response needs to be sent to U. By referring to its log file and all the recorded messages, the HNode can determine the current status of X. The existence of only P_X means that the master just

receives X from U. The existences of only P_X and I_X show that X has just been initiated. The existences of P_X, I_X, E_X^1, E_X^2, …, and E_X^G implies that X's first G phases have been finished. Hence, when taking over for the master, the HNode can continue X, rather than requiring to reperform X from the very beginning.

On the other hand, after receiving transient metadata, denoted by TM, from a worker and updating its in-memory state, the master forwards TM to the HNode without generating a log record since transient metadata as mentioned above is often updated frequently. On receiving TM, the HNode accordingly updates its in-memory state without returning a message to the master. The purpose is to reduce the master's burden and network overhead. In this way, the HNode can maintain all transient metadata of the master and does not need to spend a lot of time to reconstruct its in-memory state when taking over for the master.

```
The mutual life monitoring algorithm:    /* It is performed by receiver R. */
Input: heartbeats from sender S;
Output: a warm-up or takeover decision;
Procedure:
1:   Let C_yes=C_no=0; up = false;  /* up = false means that R has not requested the WNode to warm up.*/
2:   While a heartbeat period times out {
3:     If R has received a fresh heartbeat from S during the heartbeat period {
4:         C_yes = C_yes + 1;  C_no = 0;
5:        If C_yes = th_H and up = true {
6:            Request the WNode to sleep; up = false; }}
7:     Else{
8:         C_no = C_no + 1; C_yes = 0;
9:        If C_no = th_L and up = false {
10:           Claim itself as a commander; request the WNode to warm up; up = true;}
11:       Else if C_no = th_H {  /* Now up must be true. */
12:           If R is the HNode
13:               Take over for the master;
14:           Request the WNode to take over for the HNode; stop;}}}
```

Fig. 3. The mutual life monitoring algorithm in which th_L and th_H are two predefined thresholds, $0 < th_L < th_H$.

3.3 Mutual life monitoring algorithm

Based on a push-based failure detection technique [11], the mutual life monitoring algorithm is designed for the master and HNode to promptly detect each other's failure. Once a failure is detected, the live one starts the corresponding takeover processes and requests the WNode to warm itself up so as to accelerate the process of taking over for the HNode. In this way, the services of the master and HNode can be continued, and the extra monitor node required by the warm-standby-based schemes can be unemployed.

In this algorithm, the master and HNode mutually send a heartbeat to each other through a reliable transmission protocol, e.g., TCP, every predefined heartbeat period. Assume that at least one network link between arbitrary two of the master, HNode, and WNode is always available. Hence, if one of the master and HNode, called receiver *R*, cannot receive heartbeats from the other, called sender *S*, in a heartbeat period, the reason must be one of the three situations: *S* fails, *S* is busy, or the link between *R* and *S* is congested.

The algorithm is presented in Fig. 3. Let C_{yes} and C_{no} be the counters for counting consecutively arriving and absent heartbeats, respectively. Let up be a boolean variable indicating whether *R* has requested the WNode to warm up or not. Whenever a heartbeat period times out, if *R* has received a fresh heartbeat from *S*, then C_{yes} is increased by 1, and C_{no} is reset to zero. Note that a fresh heartbeat means that $\alpha > m$ where α is the sequence number conveyed in the heartbeat, and m is the maximum heartbeat sequence number that *R* has ever received from *S*. Conversely, if *R* receives no fresh heartbeat from *S*, then C_{no} is increased by 1, and C_{yes} is reset to zero. With this algorithm, when *S* fails, *R* will not misjudge that *S* is still alive if it receives an unfresh heartbeat from *S*.

Let th_L and th_H are two predefined thresholds, $0 < th_L < th_H$. If $C_{no} = th_L$ and up = false (see line 9 of Fig. 3), R suspects that *S* has failed, consequently claiming itself as a commander, requesting the WNode

to warm up, and setting $up = true$. This enables the WNode to early warm itself up and shorten its takeover time. If C_{no} further reaches th_H (see line 11), R claims that S has failed. Depending on the role of R, a different action is taken. If R is the master, it informs the WNode, which is now warming up or has finished warming itself up, to take over for the HNode. If R is the HNode, it takes over for the master and requests the WNode to act as the HNode. On the other hand, if $C_{yes} = th_H$ and $up = \text{true}$ (see line 5), R assumes that S is still operational and requests the WNode to sleep again.

By employing C_{yes}, C_{no}, and up, and limiting $C_{yes} = th_H$ (rather than 1), we can avoid having R continuously requesting the WNode to switch itself between the warm-up and sleep modes in adjacent heartbeat periods, consequently reducing the failure rate of the WNode and saving energy consumption.

3.4 Adaptive warm-up mechanism

The purpose of the adaptive warm-up mechanism is enabling the WNode to adapt itself to the current system status and improve its takeover performance. It consists of an *autonomous warm-up approach* and a *requested warm-up approach*. In the former, the WNode periodically and autonomously warms itself up by requesting metadata from the HNode, rather than from the master, so as not to degrade the master's performance. In the latter, the WNode warms itself up whenever being requested by a commander, which might be either the master or the HNode. Table 1 summaries the two approaches.

When the master and HNode are unstable and/or the link between them is congested, the WNode may be requested to warm up by both of them simultaneously and may receive duplicate E-records from them. Hence, on receiving a log record, the WNode compares it with all the log records that it has ever received to determine whether to discard the record or use it to update its state.

Let M_W, M_H, and M_C be the largest record numbers of the log records currently collected in the log files of the WNode, HNode, and the commander, respectively. The autonomous warm-up approach works as follows.

Step 1) The WNode wakes up when its timer, initially set to T_{warmup}, expires. Then it sends M_W to the HNode to request those log records that it lacks.

Step 2) Upon receiving M_W, the HNode replies M_H to the WNode. After that, it retrieves log record L_r from its log file, computes L_r's hash value H_r, and sends a record-hash message $\{L_r, H_r\}$ to the WNode where r is the record number of L_r and $M_W < r \leq M_H$.

Step 3) Upon receiving $\{L_r, H_r\}$, the WNode compares H_r with all the hash values previously stored in its hash pool in increasing order of r. If H_r exists, implying that L_r is duplicate, then the WNode discards this message. Otherwise, the WNode inserts L_r into its log file and stores H_r in the pool. If L_r is an E-record, then WNode further updates its state with the P-meta conveyed in L_r.

Step 4) The WNode sleeps.

On the other hand, the requested warm-up approach works as follows (i.e., the details of step 10 in Fig. 3).

Step 1) Upon receiving a warm-up request from a commander, the WNode

1-1) wakes up and resets its timer to T_{warmup} to inhibit the occurrence of the autonomous warm-up process. Note that if the autonomous warm-up process is running, the WNode immediately stops it.

1-2) The WNode sends M_W to the commander to request the log records that it lacks and all the transient metadata.

Step 2) Upon receiving the message, the commander continues sending $\{L_z, H_z\}$ and its transient metadata to the WNode until it requests the WNode to go to sleep (i.e., step 6 in Fig. 3) or asks the WNode to take over for the HNode (i.e., step 14 in Fig. 3). In the former case, the commander sends a sleep-request message {"sleep", M_C} to the WNode. For the latter case, the commander sends a takeover-request message {"takeover", M_C} to the WNode, implying that $z = M_W + 1, M_W + 2, \ldots, M_C$.

Step 3) On receiving a message m,

Case 1: If $m = \{L_z, H_z\}$, the WNode processes $\{L_z, H_z\}$ as the step 3 of the autonomous warm-up process.

Case 2: If m is transient metadata, the WNode accordingly updates its in-memory state.

Case 3: If m is a sleep-request message, the WNode sleeps after processing each received $\{L_z, H_z\}$ until $z = M_C$.

Case 4: If m is a takeover-request message, the WNode immediately acts as the HNode. The detailed takeover process will be described later.

By using the adaptive warm-up mechanism, the WNode not only can regularly back up the log records of the master from the HNode during the normal execution of the master, but also can adaptively warm itself up upon receiving a warm-up request from a commander when the master or HNode behaves abnormally. Consequently, the required takeover time can be shortened. Note that the WNode does not back up the transient metadata of the master server since transient metadata is updated frequently.

Table 1. The adaptive warm-up mechanism

Approach	Autonomous warm-up	Requested warm-up
Trigger	When T_{warmup} expires	When the WNode is requested by a commander
Message, Destination	M_W, HNode	M_W, the commander
Requested data	The log records that the WNode lacks	The log records that the WNode lacks and all the commander's transient metadata
Stop conditions	Condition 1: When finishing processing $\{L_{M_W+1}, H_{M_W+1}\}$, $\{L_{M_W+2}, H_{M_W+2}\}$, …, and $\{L_{M_H}, H_{M_H}\}$	Condition 1: When receiving a sleep-request message from the commander and finishing processing $\{L_{M_W+1}, H_{M_W+1}\}$, $\{L_{M_W+2}, H_{M_W+2}\}$, …, and $\{L_{M_C}, H_{M_C}\}$
	Condition 2: When being requested to warm up by a commander	Condition 2: On receiving a takeover-request message from the commander
Subsequent action	For condition 1, the WNode sleeps again.	For condition 1, the WNode sleeps again.
	For condition 2, the WNode stops receiving all messages sent by the HNode. After finishing processing all the received record-hash messages, the WNode starts the requested warm-up process.	For condition 2, the WNode takes over for the HNode.

3.5 Takeover processes

Let IP_M, IP_H, and IP_W (MAC_M, MAC_H, and MAC_W) be respectively the master server's, HNode's, and WNode's IP (MAC) addresses. The process for the WNode to take over for the HNode is as follows (i.e., the details of step 14 in Fig. 3).

Step 1) The WNode changes its IP address to IP_H and issues a gratuitous ARP reply message [28] to inform all the related layer-3 switches to replace <MAC_W, IP_W> recorded in their ARP tables with <MAC_W, IP_H>.

Step 2) The WNode sends a takeover-ready message to the master and starts all the HNode's functions, including synchronizing itself with the master and executing the mutual life monitoring algorithm to detect the master's heartbeats.

Note that it is possible that when the WNode is asked to take over for the HNode, it is in the middle of its requested warm-up process. In this case, the WNode keeps updating its in-memory state with the transient metadata sent by the current master, but it buffers each received log record $L_{z'}$ in its memory if $z' > M_C$. When the requested warm-up process is completed, all buffered log records will then be stored in the WNode's log file and sequentially used to update the WNode's state.

On the other hand, the process for the HNode to take over for the master is as follows (i.e., the details of step 13 in Fig. 3).

Step 1) The HNode changes its IP address to IP_M and informs all the related layer-3 switches to update their ARP tables.

Step 2) The HNode starts acting as the master and uses each prior log record to find the corresponding initial log record and E-records. If the prior log record of a persistent update operation X exists, but the corresponding initial log record does not, implying that the master has not initiated X, then the HNode re-performs X from the very beginning. If E_X^1, E_X^2, …, and E_X^i exist, the HNode performs X from the $(i+1)$-th phase to the end, $1 \le i < n$. If all E-records of X exist, the HNode does nothing since X has been finished. After that, it starts processing read/update requests submitted by clients/workers.

4. RELIABILITY ANALYSIS

This section analyzes the PAReS with the NR, HSO, A-WSO, and WSO, in terms of their job completion reliabilities, i.e., the reliability with which each of these schemes can complete a set of MapReduce jobs. To fairly compare these schemes, we assume that the HSO, A-WSO, and WSO individually employ two standby nodes for the maser, and all nodes' failure rates in their sleep (awake) modes are constant and identical, denoted by λ_s (λ_a), regardless of which scheme is analyzed. Let λ_s and λ_a individually follow a Poisson distribution, $\lambda_a > \lambda_s$. To simplify the analysis of this study, only node failures are considered. Other faults, such as network failures, will be considered in our future research.

Let $\mathcal{J}$ be a batch of MapReduce jobs submitted by users, and T be the total execution time of $\mathcal{J}$. From JobTracker's viewpoint, the execution of $\mathcal{J}$ can be divided into two phases. The first is the initialization phase, which starts from the moment when JobTracker receives $\mathcal{J}$ from the users and ends when JobTracker finishes the initialization of $\mathcal{J}$. The second is the MapReduce phase (MR-phase for short), which starts when the initialization phase finishes and ends when $\mathcal{J}$ is all completed. That is,

$$T = T_I + T_{MR} \quad (1)$$

where T_I and T_{MR} are the durations of the initialization phase and MR-phase, respectively. In general, T_I is very short, e.g., ranging from several milliseconds to several seconds, implying that

$$T \approx T_{MR} \quad (2)$$

Let $N_{\mathcal{S},1}$ and $N_{\mathcal{S},2}$ be the two standby nodes that scheme $\mathcal{S}$ provides for the master where $\mathcal{S} \in$ {PAReS, HSO, A-WSO, WSO}. Without loss of generality, we further assume that the failure sequence is the master, $N_{\mathcal{S},1}$, and then $N_{\mathcal{S},2}$. When the master fails, $N_{\mathcal{S},1}$ takes over for it. When $N_{\mathcal{S},1}$ crashes, $N_{\mathcal{S},2}$ takes over for it. Hence, if all jobs of $\mathcal{J}$ is completed before $\mathcal{S}$ fails, they must be finished by the master, $N_{\mathcal{S},1}$, or $N_{\mathcal{S},2}$. Due to the fact that the initialization phase is very short, the probability that the master fails in this phase approximates zero. Hence, in the paper, we only consider the case in which the master crashes in the MR-phase.

If $\mathcal{J}$ is finished by the master, it means that the master must to be operational in T. Hence, the corresponding reliability, denoted by R_M, is

$$R_M = \exp(-\lambda_a \cdot T) \quad (3)$$

The second case is that $\mathcal{J}$ is all finished after $N_{\mathcal{S},1}$ takes over for the master, implying that the master fails while executing $\mathcal{J}$, then $N_{\mathcal{S},1}$ must be operational in three consecutive time periods T_1 ~ T_3 as shown in Fig. 4 in which T_1 is the time consumed by the master to perform $\mathcal{J}$ before it fails (implying that $T_1 < T$), T_2 is the time required by $N_{\mathcal{S},1}$ to take over for the master, and T_3 is the time during which $N_{\mathcal{S},1}$ needs to operate normally to finish the unfinished jobs of $\mathcal{J}$, i.e.,

$$T_3 = \begin{cases} T_{rem} & \text{,if } \mathcal{S} = \text{PAReS or HSO} \\ T_{rem} & \text{, if } \mathcal{S} = \text{A-WSO or WSO in the best case} \\ T & \text{, if } \mathcal{S} = \text{A-WSO or WSO in the worst case} \end{cases} \quad (4)$$

in which T_{rem} is the duration of the remaining MR-phase, implying that $T_{rem} < T_{MR}$. Since $N_{PAReS,1}$ and $N_{HSO,1}$ keep themselves synchronized with the master, they can continue the MR-phase after taking over for the master. Hence, for the PAReS and HSO, $T_3 = T_{rem}$. For the A-WSO and WSO, if $N_{WSO,1}$ has the latest log records about $\mathcal{J}$ (we call it the best case), $T_3 = T_{rem}$. But if $N_{WSO,1}$ has no log records about $\mathcal{J}$ (we call it the worst case), then it needs to reperfrom $\mathcal{J}$ from the very beginning, implying $T_3 = T$.

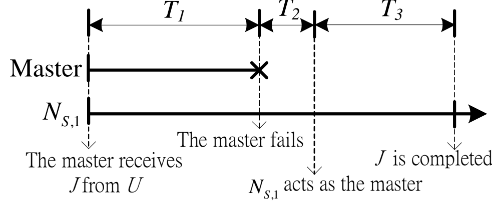


Fig. 4. When $\mathcal{J}$ is all completed after $N_{\mathcal{S},1}$ takes over for the master, $N_{\mathcal{S},1}$ should be operational in $T_1 \sim T_3$ where $\mathcal{S} \in$ {PAReS, HSO, A-WSO, WSO}.

The third case is that $\mathcal{J}$ is all completed after $N_{\mathcal{S},2}$ acts as the master, implying that both the master and $N_{\mathcal{S},1}$ fail during the execution of $\mathcal{J}$, then $N_{\mathcal{S},2}$ must work properly in five consecutive time periods T_1' ~ T_5' as shown in Fig. 5 where $T_1' = T_1$, $T_2' = T_2$, T_3' is the time during which $N_{\mathcal{S},1}$ performs $\mathcal{J}$ before it fails (implying that $T_3' < T_3$), T_4' is the time required by $N_{\mathcal{S},2}$ to take over for the master, and T_5' is the time in which $N_{\mathcal{S},2}$ needs to work normally to complete the unfinished jobs of $\mathcal{J}$. Similar to Eq. (4),

$$T_5' = \begin{cases} T_{rem}' < T_{rem} & \text{,if } \mathcal{S} = \text{PAReS or HSO} \\ T_{rem}' < T_{rem} & \text{, if } \mathcal{S} = \text{A-WSO or WSO in the best case} \\ T & \text{, if } \mathcal{S} = \text{A-WSO or WSO in the worst case} \end{cases} \quad (5)$$

where T_{rem}' is the duration time of the remaining MR-phase, implying that $T_{rem}' \leq T_{rem}$.

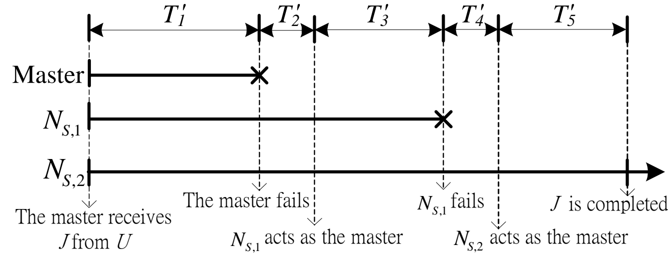


Fig. 5. When $\mathcal{J}$ is all completed after $N_{\mathcal{S},2}$ acts as the master, $N_{\mathcal{S},2}$ needs to be operational in T_1' ~ T_5'.

4.1 Job completion reliability of the NR

Since the NR does not offer any standby node for the master, when the NR is tested, the reliability with which $\mathcal{J}$ can be completed by the master, denoted by R_{NR}, is equal to R_M as shown in Eq. (3), i.e.,

$$R_{NR} = R_M = \exp(-\lambda_a \cdot T) \quad (6)$$

4.2 Job completion reliability of the HSO

In the HSO, the two hot-standby nodes (i.e., $N_{HSO,1}$ and $N_{HSO,2}$) stay awake with the master after the maste's startup. Let $R_{HSO,1}$ and $R_{HSO,2}$ be the reliabilities with which $N_{HSO,1}$ and $N_{HSO,2}$ can operate normally during $T_1 \sim T_3$ and T_1' ~ T_5', respectively. Hence,

$$R_{HSO,1} = \exp(-\lambda_a \cdot \textstyle\sum_{i=1}^{3} T_i) \quad (7)$$

and

$$R_{HSO,2} = \exp\left(-\lambda_a \cdot \textstyle\sum_{i=1}^{5} T_i'\right) \quad (8)$$

Consequently, the reliability with which $\mathcal{J}$ can be finished by the HSO, denoted by R_{HSO}, is

$$\begin{aligned} R_{HSO} &= R_M + (1 - R_M) \cdot R_{HSO,1} + (1 - R_M) \cdot \left(1 - R_{HSO,1}\right) \cdot R_{HSO,2} \\ &= R_{NR} + (1 - R_{NR}) \cdot R_{HSO,1} + (1 - R_{NR}) \cdot \left(1 - R_{HSO,1}\right) \cdot R_{HSO,2} \end{aligned} \quad (9)$$

where $(1 - R_M) \cdot R_{HSO,1}$ is the probability with which $\mathcal{J}$ is all completed when $N_{HSO,1}$ acts as the master, and $(1 - R_M) \cdot \left(1 - R_{HSO,1}\right) \cdot R_{HSO,2}$ is the probability with which $\mathcal{J}$ is all finished when $N_{HSO,2}$ acts as the master.

4.3 Job completion reliability of the A-WSO

Let ω be a warm-up percentage with which $N_{A\text{-}WSO,1}$ and $N_{A\text{-}WSO,2}$ warm themselves up before acting as the master, $0 < ω < 1$. Let $R_{A\text{-}WSO,1}$ and $R_{A\text{-}WSO,2}$ be the reliabilities with which $N_{A\text{-}WSO,1}$ and $N_{A\text{-}WSO,2}$ can operate normally during $T_1 \sim T_3$ and $T_1' \sim T_5'$, respectively. Due to the fact that $N_{A\text{-}WSO,1}$ spends $(1 - ω) \cdot T_1$ to sleep, $ω \cdot T_1$ to warm up, and $T_2 + T_3$ to act as the master,

$$R_{A\text{-}WSO,1} = \exp(-\lambda_s \cdot (1 - ω) \cdot T_1) \cdot \exp\left(-\lambda_a \cdot \left((ω \cdot T_1) + \sum_{i=2}^{3} T_i\right)\right) \quad (10)$$

On the other hand, $N_{A\text{-}WSO,2}$ consumes $(1 - ω) \cdot \sum_{i=1}^{3} T_i'$ to sleep, $ω \cdot \sum_{i=1}^{3} T_i'$ to warm up, and $\sum_{i=4}^{5} T_i'$ to act as the master. Hence,

$$R_{A\text{-}WSO,2} = \exp(-\lambda_s \cdot (1 - ω) \cdot \sum_{i=1}^{3} T_i') \cdot \exp\left(-\lambda_a \cdot \left((ω \cdot \sum_{i=1}^{3} T_i') + \sum_{i=4}^{5} T_i'\right)\right) \quad (11)$$

Recall that the A-WSO employs a monitor node to detect the master's failure so as to provide an automatic takeover process. Let R_1 and R_2 be the probabilities that the monitor node is operational in $\sum_{i=1}^{2} T_i$ and $\sum_{i=1}^{4} T_i'$, respectively, implying that $R_1 = \exp(-\lambda_a \cdot \sum_{i=1}^{2} T_i)$ and $R_2 = \exp(-\lambda_a \cdot \sum_{i=1}^{4} T_i')$. Consequently, the reliability with which the A-WSO finishes *J*, denoted by $R_{A\text{-}WSO}$, is

$$\begin{aligned} R_{A\text{-}WSO} &= R_M + (1 - R_M) \cdot R_1 \cdot R_{A\text{-}WSO,1} + (1 - R_M) \cdot \left(1 - R_{A\text{-}WSO,1}\right) \cdot R_2 \cdot R_{A\text{-}WSO,2} \\ &= R_{NR} + (1 - R_{NR}) \cdot R_1 \cdot R_{A\text{-}WSO,1} + (1 - R_{NR}) \cdot \left(1 - R_{A\text{-}WSO,1}\right) \cdot R_2 \cdot R_{A\text{-}WSO,2} \end{aligned} \quad (12)$$

4.4 Job completion reliability of the WSO

Different from the A-WSO, the WSO does not employ the monitor node. Therefore, the reliability with which the WSO finishes *J*, denoted by R_{WSO}, is

$$\begin{aligned} R_{WSO} &= R_M + (1 - R_M) \cdot R_{WSO,1} + (1 - R_M) \cdot \left(1 - R_{WSO,1}\right) \cdot R_{WSO,2} \\ &= R_{NR} + (1 - R_{NR}) \cdot R_{WSO,1} + (1 - R_{NR}) \cdot \left(1 - R_{WSO,1}\right) \cdot R_{WSO,2} \end{aligned} \quad (13)$$

4.5 Job completion reliability of the PAReS

Recall that $N_{PAReS,1}$ and $N_{PAReS,2}$ represent the HNode and WNode, respectively. Assume that the warm-up percentage of the WNode is also ω. Let R_{HNode} and R_{WNode} be the reliabilities with which the HNode and WNode can work normally during $T_1 \sim T_3$ and $T_1' \sim T_5'$, respectively. Similar to $N_{HSO,1}$, the HNode enters its hot-standby mode after the master's startup. Hence,

$$R_{HNode} = R_{HSO,1} \quad (14)$$

For the WNode, it consumes $(1 - ω) \cdot T_1'$ to sleep, $ω \cdot T_1'$ to warm up, and $\sum_{i=2}^{5} T_i'$ to act as the HNode and master. Thus

$$R_{WNode} = \exp(-\lambda_s \cdot (1 - ω) \cdot T_1') \cdot \exp\left(-\lambda_a \cdot \left(ω \cdot T_1' + \sum_{i=2}^{5} T_i'\right)\right) \quad (15)$$

Therefore, the reliability with which *J* can be finished by the PAReS, denoted by R_{PAReS}, is

$$\begin{aligned} R_{PAReS} &= R_M + (1 - R_M) \cdot R_{HNode} + (1 - R_M) \cdot (1 - R_{HNode}) \cdot R_{WNode} \\ &= R_{NR} + (1 - R_{NR}) \cdot R_{HNode} + (1 - R_{NR}) \cdot (1 - R_{HNode}) \cdot R_{WNode} \end{aligned} \quad (16)$$

4.6 Job completion reliability comparison

In this subsection, we compare the job completion reliabilities of all tested schemes under $\lambda_a = 0.0005$ per hour, $\lambda_s = 0.00001$ per hour, and $ω = 0.1$. Ten batches of jobs with total execution time $T = 2^x$ hours, $1 \le x \le 10$, were considered. Table 2 lists all required time information. We assume that the master fails when finishing one third of the MR-phase of a batch of jobs, meaning that $T_1 = \frac{T}{3} = \frac{2^x}{3}$ and $T_3 = \frac{2^{x+1}}{3}$, and $N_{S,1}$ also fails when finishing a half of the rest MR-phase, implying that $T_3' = T_5' = \frac{T_3}{2} = \frac{2^x}{3}$. Since the values of T_2, T_2', and T_4' for the PAReS, HSO, and A-WSO approximate zero hour (we will show this in Section 6.1), we set these values in Table 2 to 0. But for the WSO, its T_2, T_2', and T_4' values depend on when do system managers notice the master's failure. In order not to complicate our comparison, we also set these values to zero.

Based on Eqs. (6), (9), (12), (13), and (16), the job completion reliabilities of all schemes are illustrated in Fig. 6. Notice that when the worst cases of the A-WSO and WSO occur, both schemes are impossible to

complete those jobs, i.e., their job completion reliability are both zero. Fig. 6a shows that when T increased, the NR's job completion reliability decreased sharply, but other schemes' decreased relatively slower. The main reason is that the NR does not provide any backup node for the master. Obviously, the WSO in the best case has the highest job completion reliability since the awake times of its two warm-standby nodes are very short. However, owing to the non-adaptive backup of the WSO and the time at which the master fails is random, the best case of the WSO does not always occur. Due to employing the HNode and WNode and utilizing no monitor node that is required by the A-WSO, the PAReS's job reliabilities were higher than those of the NR, HSO, and A-WSO. This phenomenon was more significant when T was longer. Now we can conclude that the PAReS is more reliable than all the schemes, except for the WSO in the best case.

Table 2. All time information used for each tested scheme to calculate the corresponding job completion reliability (time unit: hour). Note that $x = 1,2, \ldots, 10$.

T	$T_1 = T_1'$	$T_2 = T_2'$	T_3	T_3'	T_4'	T_5'
2^x	$\frac{2^x}{3}$	0	$\frac{2^{x+1}}{3}$	$\frac{2^x}{3}$	0	$\frac{2^x}{3}$

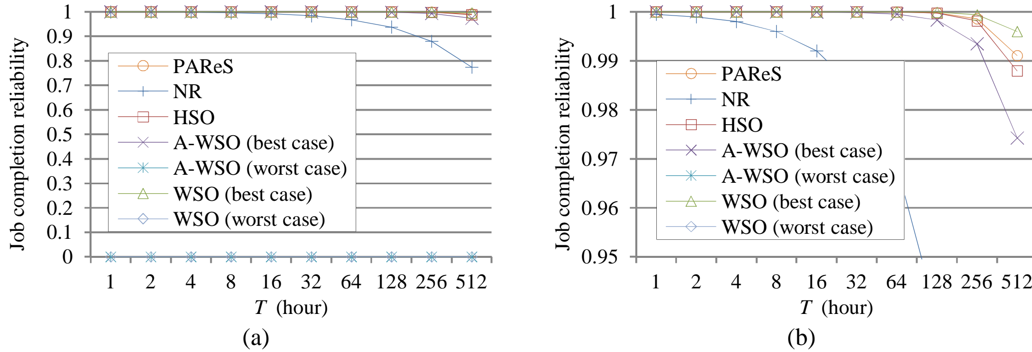


Fig. 6. Job completion reliabilities of all tested schemes on $\lambda_a = 0.0005$, $\lambda_s = 0.00001$, and $\omega = 0.1$.

5. ENERGY CONSUMPTION AND SYNCHRONIZATION COST

This section analyzes the energy consumptions and synchronization costs of all tested schemes.

5.1 Energy consumption

Assume that the master server has operated for t hours since it starts up. Let ρ_a and ρ_s be the power consumption rates of a node in its awake mode and sleep mode, respectively. Recall that ω is the warm-up percentage. When the NR is employed, the total energy consumption, denoted by E_{NR}, is equal to the total energy consumed by the master server in t, i.e.,

$$E_{NR} = t \cdot \rho_a \tag{17}$$

Due to employing two hot-standby nodes, when the HSO is utilized, the total energy consumption in t, denoted by E_{HSS}, is

$$E_{HSO} = 3 \cdot t \cdot \rho_a \tag{18}$$

When the A-WSO is utilized, the total energy consumption in t, denoted by $E_{A\text{-}WSO}$, is

$$E_{A\text{-}WSO} = 2 \cdot t \cdot \rho_a + 2 \cdot \omega \cdot t \cdot \rho_a + 2 \cdot (1 - \omega) \cdot t \cdot \rho_s \tag{19}$$

where $2 \cdot t \cdot \rho_a$ is the energy consumed by the master and monitor nodes, $2 \cdot \omega \cdot t \cdot \rho_a$ is the energy consumed by the two warm-standby nodes to back up the master's log records, and $2 \cdot (1 - \omega) \cdot t \cdot \rho_s$ the energy consumed by the two warm-standby nodes in their sleep mode.

Owing to utilizing no monitoring node, the total energy consumed by the WSO in t, denoted by E_{WSO}, is only

$$E_{WSO} = t \cdot \rho_a + 2 \cdot \omega \cdot t \cdot \rho_a + 2 \cdot (1 - \omega) \cdot t \cdot \rho_s \tag{20}$$

For the PAReS, its total energy consumption in t, denoted by E_{PAReS}, is

$$E_{PAReS} = 2 \cdot t \cdot \rho_a + \omega \cdot t \cdot \rho_a + (1 - \omega) \cdot t \cdot \rho_s \tag{21}$$

where $2 \cdot t \cdot \rho_a$ is the energy consumed by the master and the HNode, $\omega \cdot t \cdot \rho_a$ is the energy consumed by the WNode to warm up, and $(1-\omega) \cdot t \cdot \rho_s$ is the energy consumed by the WNode to sleep.

Let $\rho_a = 0.3$ kW, $\rho_s = 0.001$ kW, and $\omega = 0.1$ per hour. Fig. 7 illustrates the energy consumptions of all schemes when t ranges from 100 hours to 1000 hours. Unavoidably, the PAReS consumes more energy than the NR (approximately 2.1 times higher) and WSO (approximately 1.74 times higher). But its energy consumption is lower than those of the HSO and A-WSO. The results show that the energy consumption of the PAReS is only 70.1% of that of the HSO and 95.33% of that of the A-WSO.

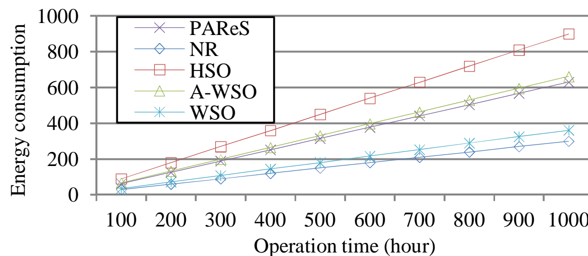


Fig. 7. The energy consumption of the all tested schemes.

5.2 Synchronization cost

Table 3 summaries all schemes' synchronization costs. Apparently, the NR has zero synchronization cost since this scheme does not synchronize the master's state with any node. For each one-phase persistent update operation, the master in the PAReS needs to generate three log records (i.e., a prior log record, an initial log record, and an E-record shown in Fig. 2). But the master in the HSO, A-WSO, and WSO only needs to generate two log records (i.e., an initial log record and a finish log record). Hence, the number of log records in the PAReS is 1.5 times those in the HSO, A-WSO, and WSO.

When the PAReS is used to perform an one-phase persistent update operation, the number of messages used by the master and the HNode to synchronize the corresponding prior log record, initial log record, and E-record are 2, 2, and 4, respectively (see Fig. 2). Hence, the total number of the messages is 8. When the HSO is employed, the number of messages used by the master and the two hot-standby nodes to synchronize the initial log record and finish record are individually 4. Thus, the total number of the messages is also 8. In addition, in the PAReS, the total number of messages sent by the master is only 3 (i.e., a prior log record, an initial log record, and an E-record to the HNode). But in the HSO, the total number of messages sent by the master is 4 (i.e., two initial and two finishes log records to the two hot-standby nodes), consequently the synchronization overhead caused by the PAReS on the master is not higher than that caused by the HSO. When the A-WSO and WSO are individually employed, no synchronization messages are transferred between the master and the two warm-standby nodes. Hence, similar to the NR, the two schemes have low synchronization cost.

For each record of transient metadata, the master in the PAReS forwards it to the HNode, but the master in the other schemes does not send any message to their standby nodes. Although the transient metadata forwarding increase network traffic, it dramatically shorten service downtime without generating significant impact on the master's performance.

Table 3. The synchronization costs of all tested schemes

Metrics \ Scheme	PAReS	NR	HSO	A-WSO	WSO
# of log records generated by the master for each persistent update operation	3	0	2	2	2
# of transmitted messages used to synchronize a persistent update operation	8	0	8	0	0
# of messages sent by the master during the synchronization of a persistent update operation	3	0	4	0	0
# of transmitted messages used to synchronize a record of transient metadata	1	0	0	0	0

6. SERVICE QUALITY EVALUATION

In this section, three experiments were presented. The first evaluated service downtimes. The second analyzed the number of client requests impacted by the master server's failure. Note that the NR and WSO schemes were not tested in the two experiments since the former does not employ any standby node and the latter does not offer automatic takeover. The third studied the WNode's takeover performance.

All experiments were conducted in a private cloud system at TungHai University, Taiwan. This system comprises fifteen nodes with 1 Gbps Ethernet network. Three nodes individually run Ubuntu 11.04 with an AMD Opteron(tm) 6172 CPU, 2 GB memory, and a 500 GB disk drive. The remaining twelve nodes individually run Ubuntu 11.04 with an AMD Athlon(tm) 2800+ CPU, 1 GB memory, and a 500 GB disk drive. Each tested scheme employed the three AMD-Opteron nodes to act as the master server and two standby nodes. No matter which scheme is tested, two AMD-Athlon nodes act as clients to submit 20 persistent-update-operation requests and 1000 read requests to the master server per second on average. The remaining AMD-Athlon nodes act as workers to send 50 records of transient metadata about data-block locations to the master every second.

All tested schemes were implemented in Java language. Note that we chose Mode 2 [9], which commits a client request after the relevant metadata is written to the master server's disk and buffered in the two hot-standby servers' memory, to implement the HSO since this mode is appropriate for a local area network that is usually employed by a MapReduce cluster. When the PAReS and A-WSO were tested, their warm-standby nodes autonomously warmed themselves up every two hours. Due to page limit and desiring to show comprehensive experiments, in the following, we only target at only one master server, i.e., NameNode. The evaluation on JobTracker can be found in our previous work [30].

6.1 Service downtime

Recall that service downtime is the time period starting when the master fails and ending when its standby node, denoted by M_{new}, takes over for it and is ready to process any request newly issued by clients. As shown in Fig. 8, this period comprises failure detection time (i.e., the time required to detect the master's failure), IP-address reconfiguration time (i.e., the time required by M_{new} to reconfigure its IP-address), and state update time (i.e., the time required by M_{new} to update its state).

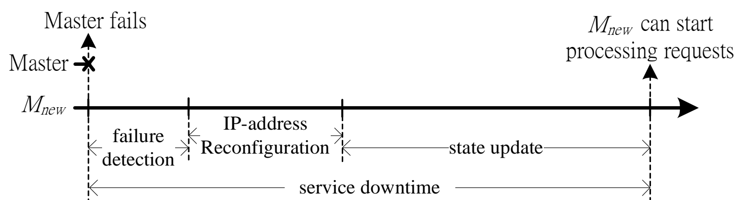


Fig. 8. Service downtime

To fairly compare the PAReS, HSO, and A-WSO, we assume that all of them employ the mutual life monitoring algorithm (see Fig. 3) to detect the master's failure. To shorten failure detection time for each tested scheme, a short heartbeat period and a small value of th_H are required for the algorithm. However, too short heartbeat period and too small th_H might increase the master's overhead and misjudge the master's failure. Hence, we conduct an experiment by disconnecting the master at random time to simulate the failure of the master server under four different heartbeat periods, including 1, 10, 100, and 100 ms. The goal is to find appropriate heartbeat period and th_H such that the master's load remains unchanged and the false positive rate is at most 0.00001. Table 4 shows that when the heartbeat period increases, the th_H value required by each tested scheme decreases since a longer heartbeat period can tolerate more heartbeat-generation delay and network transmission delay. Fig. 9 also shows that a longer heartbeat period reduce the master's CPU load, regardless of which scheme is tested. It is clearly that choosing 1000 ms as the heartbeat period is better for all tested schemes since it has insignificant impact on the master's load. Hence,

in the following experiments, we set the heartbeat period to 1000 ms for the PAReS, HSO, and A-WSO and respectively set their th_H values to 3, 3, and 2 as shown in Table 4.

Table 4. The values of th_H that satisfy the desired false positive rate.

Heartbeat period (ms)	PAReS	HSO	A-WSO
1	76	91	72
10	9	11	8
100	4	4	4
1000	3	3	2

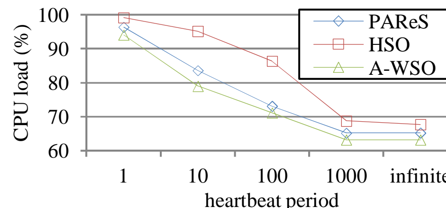


Fig. 9. The CPU load of the master when different schemes are employed.

Four failure cases as listed in Table 5 were employed to evaluate the service downtimes of the PAReS, HSO, and A-WSO. Cases 1, 2, 3, and 4 respectively represent that 5000, 10000, 50000, 100000 records of transient metadata were currently held by all workers. Each transient metadata record conveys the location information of a data block. Before the new master server can start processing requests sent by clients/workers, it needs to acquire all transient metadata held by workers to reconstruct its in-memory state. The measurement for each scheme was conducted 30 times. Table 6 lists the average service downtimes and the corresponding standard deviations (std for short) of all tested schemes. We can see that the service downtimes and stds of the PAReS were much shorter than those of the HSO and A-WSO, no matter which failure case is tested. The downtime of the PAReS were only 80.20 ($=\frac{4379.3}{5460.5}$) % to 45.85 ($=\frac{4379.3}{9550.8}$) % of those of the HSO, and 8.31 ($=\frac{4379.3}{52725.1}$) % to 7.70 ($=\frac{4379.3}{56855.2}$) % of those of the A-WSO, indicating that the PAReS can effectively shorten service downtime and provide a faster takeover. The key reason is that the PAReS's HNode always keeps receiving persistent metadata and transient metadata from the master server, i.e., the required state reconstruction time is zero.

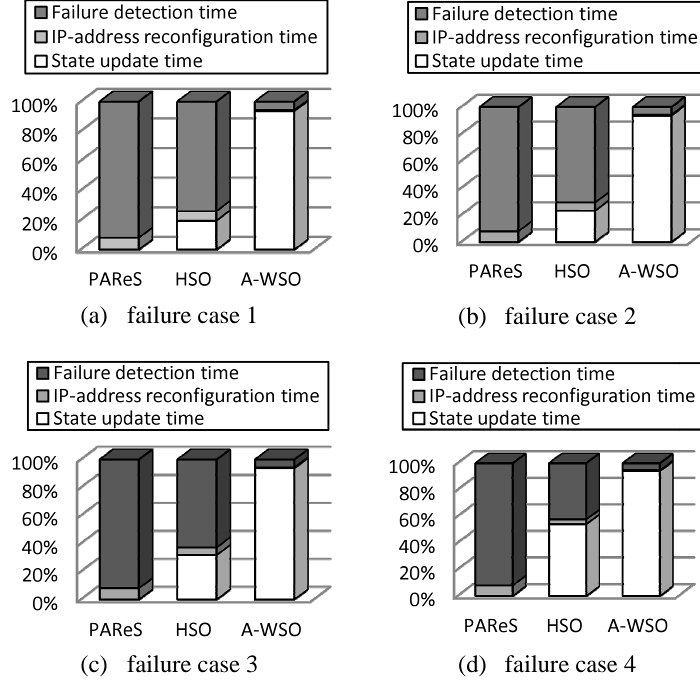
Due to holding no transient metadata, the downtime of the HSO increased as the amount of the transient metadata increased. Among all tested schemes, the A-WSO has the longest service downtime since the warm-standby node does not keep itself up with the state of the master all the time. Hence, when the warm-standby node takes over for the master, it needs to update its state with all the required log records and gather transient metadata from all workers to reconstruct its in-memory state. Fig. 10 shows that the A-WSO spends almost its service downtime in reconstructing its state.

Table 5. The failure cases considered in the experiments

Failure case	The number of Transient metadata records held by all workers
1	5000
2	10000
3	50000
4	100000

Table 6. The average service downtimes and standard deviations of the PAReS, HSO, and A-WSO (time unit: ms) when the heartbeat period is 1000 ms.

Failure case	PAReS	HSO	A-WSO
	Avg/Std	Avg/Std	Avg/Std
1	4379.3/238.1	5460.5/408.1	52725.1/31339.6
2	4379.3/238.1	5708.6/576.4	53018.6/31350.3
3	4379.3/238.1	6434.4/636.0	53711.9/31314.5
4	4379.3/238.1	9550.8/1045.5	56855.2/31300.7



(a) failure case 1 (b) failure case 2

(c) failure case 3 (d) failure case 4

Fig. 10. The time ratios comprising the service downtimes of the PAReS, HSO, and A-WSO when the heartbeat period is 1000 ms.

6.2 Interrupted and dropped requests affected by the master server's failure

In this section, we analyzed how the master's failure impacts client-submitted requests when the PAReS, HSO, and A-WSO are individually employed. One subexperiment is to analyze how many persistent-update-operation requests that have been received by the master are interrupted due to the master's failure. The other subexperiment is to evaluate how many persistent-update-operation requests submitted by client are dropped during service downtime.

6.2.1. Interrupted persistent update operations

Assume that the master server uses a queue to buffer persistent-update-operation requests sent by client, and this queue is a M/M/1/N [31] queueing model with a limited queue size N, arrival rate λ, and departure rate μ, where λ and μ both follow a Poisson distribution. Let L be the average number of persistent-update-operation requests waiting in the queue plus those currently being initialized by the master server at a moment, that is,

$$L = \begin{cases} \frac{u}{1-u} - \frac{(N+1)u^{N+1}}{1-u^{N+1}} & \text{, if } u \neq 1 \\ \frac{N}{2} & \text{, if } u = 1 \end{cases} \tag{22}$$

where $u = \frac{\lambda}{\mu}$ is the resource utilization of the master server. A higher λ implies that more resources of the master server are utilized, thus making u approximately 1. Let τ be the expected time period starting when a persistent-update-operation request arrives at the queue and ending when the master server finishes recording the first related log record. For the PAReS, the log record is the prior record of the operation, but for the HSO and A-WSO, the log record refers to the initialization record of the operation. According to [31], $\tau = \frac{1}{\mu-\lambda}$. Then we have

$$\mu = \frac{1 + \tau \times \lambda}{\tau} \tag{23}$$

This experiment was performed for one hour on $N = 60000$ and $\lambda = 20$ persistent-update-operation requests per second. The average values of τ for the PAReS, HSO, and A-WSO were respectively 0.00484, 0.00446, 4.923, and 0.00447 sec. Due to employing two hot-standby nodes, the master in the HSO needs a longer τ than those in the PAReS and A-WSO. Table 7 lists the corresponding values of μ, u, and L. The number of the operations interrupted by the PAReS (i.e., 0.1) is much lower than that of the HSO, implying that the PAReS improves takeover completeness. Although the A-WSO has a small L, it does not mean that the two warm-standby nodes have backed up this record, implying in reality more than 0.09 operations will be interrupted by the A-WSO.

Table. 7 The values of μ, u, and L of the PAReS, HSO, and A-WSO when $\lambda = 20$ requests/sec. Note that $1.0968 = 1 + 0.00484 \times 20$, $99.46 = 1 + 4.923 \times 20$, and $1.0894 = 1 + 0.00447 \times 20$ based on the numerator of Eq. (23)

Scheme	μ	u	L
PAReS	$226.61\ (= \frac{1.0968}{0.00484})$	$0.088\ (= \frac{20}{226.61})$	$0.1\ (= \frac{0.088}{1-0.088})$
HSO	$20.2\ (= \frac{99.46}{4.923})$	$0.990\ (= \frac{20}{20.2})$	$99\ (= \frac{0.990}{1-0.990})$
A-WSO	$243.71\ (= \frac{1.0894}{0.00447})$	$0.082\ (= \frac{20}{243.71})$	$0.09\ (= \frac{0.082}{1-0.082})$

6.2.2. Dropped persistent update operations

Based on the results shown in Table 6, the average number of persistent-update-operation requests dropped during each scheme's service downtime can be calculated by multiplying the downtime and λ where $\lambda = 20$ persistent-update-operation requests per second. The results presented in Fig. 11 show that the PAReS outperforms the HSO and A-WSO. Only 87.59 persistent-update-operation requests were dropped during the PAReS's service downtime. This is because the PAReS has a shorter service downtime than those of the HSO and A-WSO, and the PAReS's downtime was not impacted by the four abovementioned failure cases.

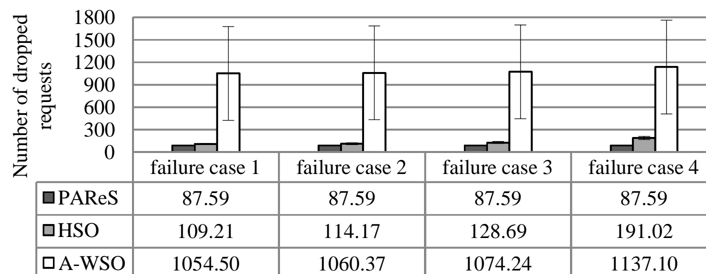


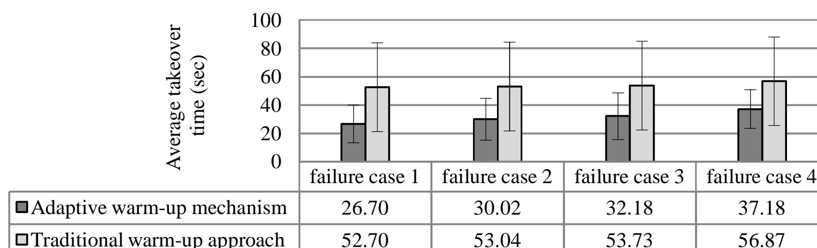
Fig. 11. The average number of persistent-update-operation requests dropped during the service downtimes of the PAReS, HSO, and A-WSO on different failure cases.

6.3 WNode takeover performance

Recall that the adaptive warm-up mechanism of the PAReS's WNode consists of the autonomous warm-up approach and the requested warm-up approach (see Section 3.5). In this experiment, we evaluate the times required by the WNode to completely act as for the HNode when the WNode individually uses the adaptive warm-up mechanism and the traditional warm-up mechanism (i.e., the autonomous warm-up approach). The goal is to show how the adaptive warm-up mechanism improves the takeover performance of the traditional warm-up approach.

Assume that in the two tested mechanisms, the WNode periodically warms itself up every two hours. Let heartbeat period be 1000 ms, and then based on Table 4 the th_H value for both mechanisms are 3. In addition, let $th_L = 2\ (= \lceil \frac{th_H}{2} \rceil)$ in the adaptive warm-up mechanism. The abovementioned four failure cases are also considered in this experiment. Fig. 12 illustrates the average results, which shows that the takeover time of the WNode in the adaptive mechanism were only 50.66 $(= \frac{26.70}{52.70})$ % to 65.38 $(= \frac{37.18}{56.87})$ % of those in

the traditional mechanism. The first reason is that the WNode warmed itself up when $C_{no} = 2$ and $up =$ false, rather than when $C_{no} = th_H = 3$. The second reason is that the master server either failed suddenly or failed after behaving unstably for a period of time. Fig. 12 also shows that the standard deviations of the takeover times in both mechanisms were large since the time points at which the master server failed were random. Hence, the times required by the WNode to completely act as the HNode varied dramatically. Nevertheless, the adaptive mechanism leads to a smaller standard deviation than the traditional one.



	failure case 1	failure case 2	failure case 3	failure case 4
Adaptive warm-up mechanism	26.70	30.02	32.18	37.18
Traditional warm-up approach	52.70	53.04	53.73	56.87

Fig. 12. Average time required by the WNode to take over for the master when the adaptive warm-up mechanism and the traditional warm-up mechanism are employed, respectively.

7. CONCLUSIONS AND FUTURE WORK

In this study, we have introduced the PAReS for MapReduce master servers to improve their job completion reliability and service quality, including short service downtime and seamless takeover, at acceptable energy consumption level and synchronization cost. We have also conducted formal analyses and extensive experiments to compare the PAReS with four state-of-the-art schemes, including the NR, HSO, A-WSO, and WSO, in terms of their job completion reliabilities, energy consumptions, synchronization costs, service downtimes, and impacts on client-submitted requests.

The PAReS dramatically improves the job completion reliability, service downtime, and the impacts on client-submitted requests of the NR, even though the NR due to employing no standby node for the master has lower energy consumption and synchronization cost than the PAReS.

Owing to employing the HNode and WNode and requiring no extra monitor node, the PAReS has higher job completion reliability, lower energy consumption, shorter service downtime, and less client-requests impact when compared the HSO and A-WSO. Unavoidably, the PAReS has higher synchronization cost than the A-WSO, but the synchronization cost of the PAReS is acceptable as compared with that of the HSO.

Due to adopting a manual takeover process, the service downtime and client-request impact of the WSO depends on how soon do system manages detect the master's failure and use the WSO to take over for the failed master. In addition, the job completion reliability of the WSO depends on whether the WSO's warm-standby node has the log records of the corresponding jobs. If yes, these jobs can be continue by the WSO. Otherwise, these jobs cannot continue. Even though the WSO has lower energy consumption than the PAReS, the abovementioned shortcomings make the WSO unable to offer good service quality.

In the future, we plan to study JobTracker and NameNode failures caused by other reasons, such as network failure, and take worker failure into consideration to build a more reliable MapReduce environment. These constitute our future research.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communication of the ACM* 51(1) (2008), pp.107–113.
[2] Apache Hadoop, http://hadoop.apache.org. Accessed 12 March 2013
[3] Gridgain, http://www.gridgain.com. Accessed 15 March 2013
[4] MapSharp, http://mapsharp.codeplex.com. Accessed 15 March 2013
[5] Skynet, http://skynet.rubyforge.org. Accessed 15 March 2013
[6] T. White, "Hadoop: The definitive guide," O'Reilly Media, Yahoo! Press, June 5, 2009.
[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in: *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies*, , 2010, pp.1–10.

[8] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J.M. Hellerstein, and R.C Sears, “BOOM: Data-Centric Programming in the Datacenter,” *Technical Report UCB/EECS-2009-113*, EECS Department, University of California, Berkeley, Jul 2009.
[9] F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, “Hadoop High Availability through Metadata Replication,” in: *Proceedings of the first international workshop on Cloud data management*, ACM, 2009, pp. 37–44.
[10] P. Hunt, M. Konar, F. Junqueira, and B. Reed, “Zookeeper: Wait-free Coordination for Internet-scale Systems,” in: *Proceedings of the USENIX Annual Technical Conference*, 2010, pp. 145–158.
[11] P. Felber, X. D´efago, R. Guerraoui, and P. Oser, “Failure Detectors as First Class Objects,” in: *Proceedings of the 9th IEEE International Symposium on Distributed Objects and Applications*, 1999. pp. 132–141.
[12] O.G. Loques and J. Kramer, “Flexible Fault Tolerance for Distributed Computer Systems,” in: *IEE Proceedings-E on Computers and Digital Techniques* 133(6), 1986, pp. 319–337.
[13] M.L. Shooman, “Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design,” New York: John Wiley & Sons Inc., 2002.
[14] S.V. Amari and G. Dill, “A New Method for Reliability Analysis of Standby Systems,” in: *Proceedings of Annual Reliability & Maintainability Symposium*, 2009, pp 417–422.
[15] F.Y. Leu, C.T. Yang, and F.C. Jiang, “Improving Reliability of a Heterogeneous Grid-based Intrusion Detection Platform using Levels of Redundancies,” Future Generation Computer Systems 26(4), 2010, pp.554–568.
[16] Y. Du and H. Yu, “Paratus: Instantaneous Failover via Virtual Machine Replication,” in: *Proceedings of 8th International Conference on Grid and Cooperative Computing*, 2009, pp.307–312.
[17] C. Engelmann, H.H. Ong, and S.L. Scott, “The Case for Modular Redundancy in Large-scale High Performance Computing Systems,” in: *Proceedings of the 27th IASTED International Conference on Parallel and Distributed Computing and Networks*, 2009, pp. 189–194.
[18] L. Huang and Q. Xu, “Lifetime Reliability for Load-Sharing Redundant Systems with Arbitrary Failure Distributions,” *IEEE Trans. on Reliability*, Volume 59, Issue 2, pp. 319–330, 2010.
[19] D. Bri`ere and P. Traverse, “AIRBUS A320/A330/A340 Electrical Flight Controls – A family of fault tolerant systems,” in: *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, 1993, pp. 616–623.
[20] Y.C. Yeh, “Triple-triple Redundant 777 Primary Flight Computer,” in: *Proceedings of the IEEE Aerospace Applications Conference*, volume 1,1996, pp. 293–307.
[21] M. Pignol, “How to Cope with SEU/SET at System Level,” in: *Proceedings of the IEEE International On-Line Testing Symposium*, 2005, pp. 315–318.
[22] F. Marozzo, D. Talia, and P. Trunfio, “A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments, Cloud Computing, 2010, pp.113–125.
[23] X. He, L. Ou, C. Engelmann, X. Chen, and S.L. Scott, “Symmetric Active/Active Metadata Service for High Availability Parallel File Systems,” *Journal of Parallel and Distributed Computing* 69(12), 2009, pp. 961–973.
[24] Z. Chen, J. Xiong, and D. Meng, “Replication-based Highly Available Metadata Management for Cluster File Systems,” in: *Proceedings of the IEEE International Conference on Cluster Computing*, 2010, pp. 292–301.
[25] P. Alsberg and J. Day, “A Principle for Resilient Sharing of Distributed Resources,” in: *Proceedings of the 2nd Internal Conference on Software Engineering*, 1976, pp. 627–644.
[26] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, “The Primary-Backup Approach,” in: *Distributed systems 2*, 1993, pp. 199–216.
[27] R. van Renesse and F.B. Schneider, “Chain Replication for Supporting High Throughput and Availability,” in: *Sixth Symposium on Operating Systems Design and Implementation*, 2004, pp. 91–104.
[28] Gratuitous ARP, http://wiki.wireshark.org/Gratuitous_ARP. Accessed 20 April 2013
[29] J. Wan, M. Liu, X. Hu, Z. Ren, J. Zhang, W. Shi, and W. Wu, “Dual-JT: Toward the High Availability of JobTracker in Hadoop,” in: *IEEE 4th International Conference on Cloud Computing Technology and Science*, pp. 263–268. 2012.
[30] J.C. Lin, F.Y. Leu, Y.-p. Chen, “ReHRS: A Hybrid Redundant System for Improving MapReduce Reliability and Availability”, to be appeared on *Modelling and Processing for Next Generation Big Data Technologies and Applications, Springer Series in Modeling and Optimization in Science & Technology.*