International Journal of Trend in Research and Development, Volume 2(3), ISSN: 2394-9333
www.ijtrd.com

Improvement in the Performance of Byzantine Fault Tolerant in Hadoop

[1]K.Ganesan and [2]K. Krishneswari,
[1]M.E(Computer Science & Engineering),
[2]Professor and Head, Dept Of Computer Science & Engineering,
[1,2]Tamilnadu College of Engineering, Coimbatore, India.

Abstract:

MapReduce is emerging as an important programming model for large-scale data-parallel applications such as web indexing, data mining, and scientific simulation. MapReduce runtimes like Hadoop tolerates only crash faults, not arbitrary or Byzantine faults. Byzantine fault tolerance algorithm in MapReduce typically requires 3f+1 servers to tolerate f Byzantine Servers, which involves considerable costs in hardware, software, and administration. By achieving arbitrary fault tolerance using proposed Byzantine Fault Tolerance (BFT) MapReduce algorithms, which improve previous algorithms in terms of several metrics. First, design a framework requires only 2f + 1 replicas, instead of the usual 3f + 1. Second, improve the performance of framework with help of non-speculative and speculative algorithms. An important aspect in terms of BFT MapReduce algorithm executes the job with acceptable cost for many critical applications.

Keywords: Hadoop, MapReduce, Byzantine Faults

1.INTRODUCTION

Hadoop is a most popular open-source software framework implemented using Java and is designed to be used on large distributed systems. Hadoop is a project of the Apache Software Foundation and is a very popular software tool due, in part, to it being open-source. Yahoo! has contributed to about 80% of the main core of Hadoop, but many other large technology organization have used or are currently using Hadoop, such as, Facebook, Twitter, LinkedIn and others. The Hadoop framework is comprised of many different projects, but two of the main ones are the Hadoop Distributed File System (HDFS) and MapReduce.

The Hadoop Distributed File System (HDFS) is the file system component of the Hadoop framework. HDFS is designed and optimized to store data over a large amount of low-cost hardware in a distributed fashion. HDFS is comparable to Google's BigTable and designed for a large amount of big data files. A typical data file stored using HDFS could range from terabytes to zeta bytes in size. It support millions of files and can scale to hundreds of nodes. HDFS stores file system metadata and application data separately. The HDFS metadata is stored on dedicated server called Name node and application data are stored on other node called Data node. The Communication in HDFS among all nodes in the system is done by using a TCP-based protocol.

MapReduce is a parallel programming model for processing large amounts of metadata on cluster computers with unreliable and weak communication links [3]. MapReduce is based on the scale-out principle, which involves clustering a large number of desktop computers. The main point of using MapReduce is to move computations to data nodes, rather than bring data to computation nodes, and thus fully utilize the advantage of data locality. The code that divides work, exerts control, and merges output in MapReduce is entirely hidden from the application user inside the framework. In fact, most of the parallel applications can be implemented in MapReduce as long as synchronized and shared global states are not required. MapReduce allows the computation to be done in two stages: the map stage and then the reduce stage. The data are split sets of key–value pairs and their instances are processed in parallel by the map stage, with a parallel number that matches the node number dedicated as slaves. This process generates intermediate key–value pairs that are temporary and can later be directed to reduce stages. Within map stages or reduce stages, the processing is conducted in parallel. The map and reduce stages occur in a sequential manner by which the reduce stage starts when the map stages finishes.

Fault Tolerance [2] is the ability of a system to continue to function correctly and not lose data

even after some components of that system have failed. It is complex to achieve fault tolerant because there are many physical circumstances that just cannot be planned for, but goal of fault tolerance it is important to eliminate single point of failure, where are elements of the system, that when they fail, they can bring down the whole system. One of the main goals of Hadoop and HDFS is to be highly fault tolerant. When considering that thousands of computer components and hundreds of networks devices such as switches, routers and power units that are involved in these large distributed systems, it cause failures to be frequent. Hadoop and HDFS center its fault tolerance on data redundancy, which is to replicate data so that if one replica is lost then there are backup copies.

A recent era long study of Dynamic Random Access Memory (DRAM) errors in a large number of servers in Google Datacenter [9], concluded that these errors are more prevalent than previously believed, with more than 8% Dual In-Line Memory Module (DIMM) affected by yearly, even if protected by Error Correcting Code (ECC). The Fault tolerance mechanisms of the original MapReduce and Hadoop cannot deal with such arbitrary or Byzantine faults, even if considering only accidental faults, not malicious faults. These faults cannot be detected using file checksums, so they can silently corrupt the output of any map or reduce task, corrupting the result of MapReduce job.

Exploring a new framework to leverages the weakness of Hadoop that execute the task in two modes such as Speculative and Non-Speculative in Job Tracker. Additionally, differ from existing approach that use around twice more resources instead of three times more of alternative solutions. Byzantine fault tolerance framework allows reduced number of faulty replica to attain improve its ability to mask software errors with acceptable cost.

1.1 HADOOP AND MAPREDUCE

Hadoop was originally developed to be an open implementation of Google MapReduce and Google File System. As the ecosystem around Hadoop has matured, a variety of tools have been developed to streamline data access, data management, security, and specialized additions for verticals and industries. Despite this large ecosystem, there are several primary uses and workloads for Hadoop that can be outlined as:

Storage: One primary component of the Hadoop ecosystem is HDFS—the Hadoop Distributed File System. The HDFS allows users to have a single addressable namespace, spread across many hundreds or thousands of servers, creating a single large file system. HDFS manages the replication of the data on this file system to ensure hardware failures do not lead to data loss. Many users will use this scalable file system as a place to store large amounts of data that is then accessed within jobs run in Hadoop or by external systems.

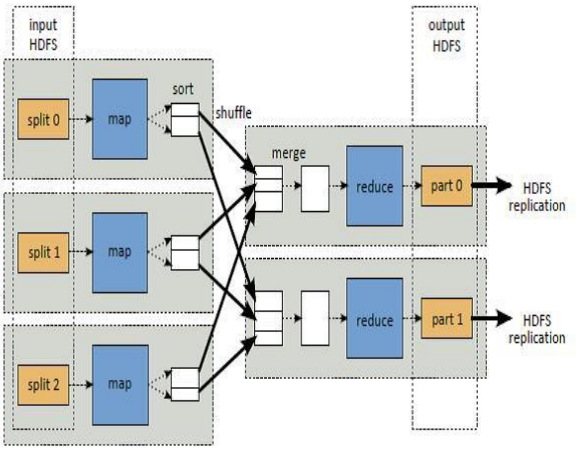
Compute: A common use of Hadoop is as a distributed compute platform for analyzing or processing large amounts of data. The compute use is characterized by the need for large numbers of CPUs and large amounts of memory to store in-process data. The Hadoop ecosystem provides the Application Programming Interfaces (APIs) necessary to distribute and track workloads as they are run on large numbers of individual machines.

Database: The Hadoop ecosystem contains components that allow the data within the HDFS to be presented in a SQL-like interface. This allows standard tools to INSERT, SELECT, and UPDATE data within the Hadoop environment, with minimal code changes to existing applications. Users will commonly employ this method for presenting data in a SQL format for easy integration with existing systems and streamlined access by users.

The MapReduce programming model computes a job in two phases programmers specify two functions, map and reduce. The input is typically large (e.g., gigabytes) and divided in files called splits. In the first phase, each split is processed by the map function that generates key-value pairs. Then, these outputs are shuffled according to their keys and passed to the reduce tasks (each reduce typically gets input from all maps) that process the again. This simple idea has been shown to be useful for many different applications.

Both the splits and the outputs of the reduce tasks are stored in a file system. Due to the typical large size of these files, Hadoop has a specific file system for this purpose, HDFS, similar to Google's Google File System (GFS). HDFS stores files in blocks of 64 MB by default. HDFS contains a name

node that manages data storage and many data nodes, typically one per server, which store the blocks. Blocks are usually replicated in a few data nodes for fault tolerance.



Computation of job in MapReduce

Users submit a job by providing the map and reduce functions, and the location of the splits in the HDFS. The processing of a job is controlled by the job tracker, which is centralized. The map and reduce tasks are executed by task trackers, which are executed in servers (e.g., one per core). Whenever possible, a map task is executed in the server storing the split it must process (locality).

Task trackers periodically send heartbeat messages to the job tracker. The missing of heartbeat messages allows the job tracker to figure out that a task stalled or failed. Using different nodes, the job tracker runs extra, speculative, tasks for those lagging behind and restarts the failed ones. Nevertheless, this model only supports crashes, not arbitrary faults.

1.2 HDFS ARCHITECTURE

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

1.3 NAME NODE AND DATA NODE

HDFS has master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux Operating System (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

1.4 THE FILE SYSTEM NAMESPACE

HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. HDFS does not yet implement user quotas or access permissions. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features. The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

1.5 DATA REPLICATION

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time. The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

1.6 THE COMMUNICATION PROTOCOLS

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the ClientProtocol with the NameNode. The DataNodes talk to the NameNode using theDataNode Protocol. A Remote Procedure Call (RPC) abstraction wraps both the ClientProtocol and the DataNode Protocol. By design, the NameNode never initiates any RPCs.Instead, it only responds to RPC requests issued by DataNodes or clients.

1.7 ROBUSTNESS

The primary objective of HDFS is to store data reliably even in the presence of failures. The three common types of failures are NameNode failures, DataNode failures and network partitions. Each DataNode sends a Heartbeat message to the NameNode periodically. A networkpartition can cause a subset of DataNodes to lose connectivity with the NameNode. The NameNode detects this condition by the absence of a Heartbeat message. The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them. Any data that was registered to a dead DataNode is not available to HDFS anymore. DataNode death may cause the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary. The necessity for re-replication may arise due to many reasons: a DataNode may become unavailable, a replica may become corrupted, a hard disk on a DataNode may fail, or the replication factor of a file may be increased.

1.8 DATA ORGANIZATION

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 64 MB. Thus, an HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.A client request to create a file does not reach the NameNode immediately. In fact, initiallythe HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it.

The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified DataNode. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. However, this degradation is acceptable because even though HDFS applications are very data intensive in nature, they are not metadata intensive. When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use. The NameNode machine is a single point of failure for an HDFS cluster. If the NameNode machine fails, manual intervention is necessary. Currently, automatic restart and failover of the NameNode software to another machine is not supported.

1.9 FILE DELETES AND UNDELETES

When a file is deleted by a user or an application, it is not immediately removed from

HDFS. Instead, HDFS first renames it to a file in the /trash directory. The file can be restored quickly as long as it remains in /trash. A file remains in /trash for a configurable amount of time. After the expiry of its life in /trash, the NameNode deletes the file from the HDFS namespace. The deletion of a file causes the blocks associated with the file to be freed. Note that there could be an appreciable time delay between the time a file is deleted by a user and the time of the corresponding increase in free space in HDFS.

2. SYSTEM ANALYSIS

2.1 PROBLEM DESCRIPTION

The Byzantine Fault Tolerance (BFT) MapReduce runtime system by Costa was proposed to tolerate system faults that corrupt the results of computation of tasks (e.g. DRAM and CPU errors or faults). A replication approach is applied over a Hadoop implementation, and the authors proposed several methods to reduce the number of replicas needed: i) deferred execution, which allows reducing the number of replicas from 3f +1 to f+1, where f is the number of errors to tolerate; ii) tentative reduce execution, which starts executing the reduce tasks when the first replicas complete their execution; iii) digest outputs, which fetches the (usually large) outputs from replicas and compare them to hashes, thus reducing the network traffic; and iv) tight storage replication that implies writing only one replica for the outputs of both map and reduce tasks. A prototype of BFT was implemented by modifying the Hadoop version 0.20.0 source code, including changes into the JobTracker class to implement the replica management using queues, and the Job in progress class to store information (into a Voting System object) about each running replica. The Heartbeat class was slightly modified to include the digest and task replica identifiers. The experimental evaluation compared the BFT system and the traditional Hadoop implementation, for the case of f=1 errors. Executions of MapReduce using BFT are about 15% larger than using Hadoop, but they provide support for single execution errors at a cost of roughly twice the consumption of resources. Similar to other fault tolerance mechanisms using replication, the proposed BFT system is more economical than using the original MapReduce runtime implemented on Hadoop.

2.2 EXISTING SYSTEM

Byzantine fault existence can be extremely difficult to design a highly reliable system. The only practical solution is to build fault tolerant framework. To achieve fault tolerance system must have one or more mechanism specifically designed for dealing with component failures (network switches, routers and power units), hard drive and machine failures.

Two of the notable existing system describes in following approaches,

- Creditability based fault tolerance
- State machine replication

2.2.1 CREDITABILITY BASED FAULT TOLERANCE

Creditability Based Fault Tolerant (CBFT) [4] working behind a traditional majority voting technique integrates with new idea of Spot-checking, Backtracking, and Backlisting mechanism. With the help of the mechanism is to estimate the creditability of results and workers as the probability of their being correct the given results. Estimated the given results to determine whether a piece of work needs to be repeatable or is creditable enough to be accepted, not only able to attain mathematically guranteeable level of correctness, but are also able to do so much smaller down than possible with traditional techniques. Finally, validate these new ideas with Monte-Carlo Simulation.

CBFT to attain their efficiency and reliable framework for MapReduce by using following techniques,

a. Majority Voting
b. Spot Checking
c. Spot Listing

A. Majority Voting

Each piece of work several times and decide which resultto accept through a vote. Majority voting can easily implement this scheme by using a modified eager scheduling work pool.The master continuouslygoes through the work entries in the work pool in round-robin fashion, until the done flags of all work entries are set. In this case, however, the done flag of each work entry is left unset until collect m matching results for that work entry, thus implementing an m-first voting scheme.

B. Spot Checking

In spot-checking, the master node does not redo all the work objects two or more times, but instead randomly gives a worker a spotter work object whose correct result is already known or will be known by checking it in some manner afterwards. Then, if a worker is caught giving a bad result, the master back-tracks through all the results received from that worker so far and invalidates all of them. The master may also blacklist the caught saboteur so that it is prevented from submitting any more bad results in the future. Because spot-checking does not involve replicating all the work objects, it has a much lower redundancy than voting.

C. Spot Listing

In Spot Listing, saboteurs are blacklisted and never allowed to return or do any more work. Unfortunately, it may not always be possible to enforce blacklisting. Although, blacklist the saboteurs based on email address, it is not too hard for a saboteur to create a new email address and volunteer as a new person. Blacklisting by IP address would not work either because many people use ISPs that give them a dynamic address that changes every time they dial up. Requiring more verifiable forms of identification such as home address and a telephone number can turn away saboteurs, but would probably turn away many well-meaning volunteers as well.In this case,errors can only come from saboteurs that survive until the end of the batch.

2.2.2 STATE MACHINE REPLICATION

The state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas [1]. A state machine consists of state variables, which encode its state, and commands, which transform its state. Each command is implemented by a deterministic program; execution of the command is atomic with respect to other commands and modifies the state variables and/or produces some output. State machine can be implemented by replicating that state machine and running a replica on each of the processors in a distributed system. Provided each replica being run by a non-faulty processor starts in the same initial state executes the same request in the same order, then each will do the same thing and produce the same output.

When processors can experience Byzantine failures, an ensemble implementing a fault tolerant state machine must have at least 2f+1 replica, and the output of the ensemble is the output produced by the majority of the replicas.

To implementing fault-tolerant state machine is to ensure that following:

Replica Coordination: All replicas receive and process the same sequence of requests.
Agreement: Every non-faulty state machine replica receives every request.
Order: Every non-faulty state machine replica processes the request it receives in the same relative order

2.2.3 LIMITATIONS OF EXISTING SYSTEM

All computation is performed in an untrusted environment, and at any time, one can expect some volunteers to sabotage.There is no guarantee that a scheduled task will produce a result in an acceptable time frame.

If only fail-stop failures are possible, *t+1* replicas are sufficient for a *t* fault tolerant system, but byzantine failure requires 3t+1 replica.

State machine approach is not directly applicable to the replication of MapReduce tasks, only to replicate the jobs, which is expensive.

Replicates everything includes task execution, map tasks input readings, communication of map tasks output and storage of map reduce outputs in 2f+1 manner.

Transferring the same data several times causing additional network traffic.

2.3 PROPOSED SYSTEM

2.3.1 OVERVIEW OF THE PROJECT

The objective of designing a new system is to improve the performance of MapReduce programming model with considerable cost over the conventional mechanisms. A traditional State machine replication mechanism replicates everything 3f+1 time includes task execution, map task outputs, and storage of reduce task outputs. To optimize an execution cost, Byzantine Fault Tolerant (BFT) MapReduce algorithm execute each job twice more resources instead of three times more of alternative solutions. An algorithm can run in two modes for handling byzantine failures in large cluster, speculative and non-speculative mode.

In speculative mode, reduce tasks start after one replica of all map finish and the non-speculative mode, f+1 replica of all map tasks have to complete successfully.

BFT MapReduce Algorithm

A simplistic solution to make MapReduce Byzantine fault-tolerant considering the maximum faulty replica that can return the same output given the same input instead of maximum number of replica used in conventional algorithm. It works behind the steps to be followed as,

The JobTracker starts 2f+1 replicas of each map task in different nodes and TaskTrackers.

The JobTracker starts also 2f+1 replicas of each reduce task. Each reduce task fetches the output from all map replicas, picks the most voted results, processes them and stores the output in HDFS. At the end, either the client or a special task must vote the outputs to pick the correct result.

A simpler but powerful idea of fault tolerant algorithm to avoid the execution cost by a set of following techniques,

A. Deferred execution

Crash faults, which happen more often, are detected using Hadoop standard heartbeats, while arbitrary faults are dealt using replication and voting. Given the expected low probability of arbitrary faults, there is no point in always executing 2f + 1 replicas to obtain the same result almost every time. Therefore, our job trackers start only f +1 replicas of map and reduce tasks. After map tasks finish, the reduce tasks check if all f + 1 replicas of every map tasks produced the same output. If some outputs do not match, more replicas are started until there are f + 1 matching replies. At the end of execution, the reduce output is also checked to see if it is necessary to launch more reduce replicas.

B. Digest outputs

In Digest output, using f+1 map outputs and f+1 reduce outputs must be matched to be considered correct. These outputs tend to be large, so it is useful to fetch only one output from some task replica and compare its digest with those of the remaining replicas. With this solution, to avoid transferring the same data several times causing additional network traffic, and just transfer data from one replica and the digests from the rest.

C. Tight storage replication

To write the output of all reduce tasks to HDFS with a replication factor of 1, instead of 3 (the default value). These are already replicating the tasks, and their outputs will be written on different locations, so no need to replicate these outputs even more. A job starts reading replicated data from HDFS, but from this point forward, the data that is saved in the HDFS by each (replicated) task is no longer replicated.

3. CONCLUSION AND FUTURE WORK

3.1 CONCLUSION

The fault tolerance mechanisms of the original MapReduce cannot deal with Byzantine faults. These faults in general cannot be detected, so they can silently corrupt the output of any map or reduce task. A novel algorithm, Byzantine Fault Tolerant (BFT) MapReduce masks these faults by executing each task more than once, comparing the outputs of these executions, and disregarding non-matching outputs.

BFT MapReduce focuses three major problems in Hadoop environment. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, a number of optimizations in proposed system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves cost. Third, redundant execution can be used to reduce the impact of slow machines, and to handle arbitrary failures and data loss.

This simple but powerful idea allows BFT MapReduce to tolerate any number of faulty task executions at the cost of one re-execution per faulty task..

3.2 FUTURE WORK

Apache Pig is a platform for analyzing large data sets that consists of high-level language for expressing data analysis programs coupled with extensibility. But it has lack of security and

performance degradation due to failover. Furthermore, decided to BFT MapReduce algorithm applied in Apache Pig Platform to overcome those difficulties.

REFERENCES

[1] Pedro Costa, Marcelo Pasin, Alysson Bessani, Miguel Correia, "On the Performance of Byzantine Fault-Tolerant MapReduce", IEEE Transactions on Dependable and Secure Computing, vol.22, no. 9, Mar. 2013

[2] P. Costa, M. Pasin, A. Bessani, and M. Correia, "Byzantine fault tolerant MapReduce: Faults are not just crashes", in Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science, 2011, pp.32–39.

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", in Proceedings of the 6th Symposium on Operating Systems Design & Implementation, Dec. 2004.

[4] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for dataintensive scientific analyses", in Proceedings of the 2008 Fourth IEEE International Conference on eScience, 2008, pp. 277–284.

[5] M. Moca, G. C. Silaghi, and G. Fedak, "Distributed results checking for MapReduce in volunteer computing", in Proceedings of the 5th Workshop on Desktop Grids and Volunteer Computing Systems, May 2011.

[6] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, andC. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems", in Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture, 2007, pp.13–24.

[7] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems", Future Generation Computer Systems, vol. 18, pp. 561–572, Mar. 2002.

[8] F. B. Schneider, "Implementing fault-tolerant service using the state machine approach: A tutorial," ACM Computing Surveys, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[9] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan and David I. August, "SWIFT: Software Implemented Fault Tolerance", in Proceedings of the Code Generation and Optimization International Symposium, Mar. 2005.

[10] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin,"Separating agreement form execution for Byzantine fault tolerant services", in Proceedings of the 19th ACM Symposium on Operating Systems Principles, Oct. 2003, pp. 253–267.

[11] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica,"Improving MapReduce performance in heterogeneous environments", in Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation, 2008, pp. 29–42.