Enhanced Failure Detection Mechanism in MapReduce

Bunjamin Memishi
PhD student - 2nd year,
Facultad de Informática, Universidad Politécnica de Madrid, Spain
Email: bmemishi@fi.upm.es

Dissertation Advisors: María S. Pérez and Gabriel Antoniu

DOCTORAL DISSERTATION COLLOQUIUM

EXTENDED ABSTRACT

***Abstract*—The popularity of MapReduce programming model has increased interest in the research community for its improvement. Among the other directions, the point of fault tolerance, concretely the failure detection issue seems to be a crucial one, but that until now has not reached its satisfying level. Motivated by this, I decided to devote my main research during this period into having a prototype system architecture of MapReduce framework with a new failure detection service, containing both analytical (theoretical) and implementation part. I am confident that this work should lead the way for further contributions in detecting failures to any NoSQL App frameworks, and cloud storage systems in general.**

***Index Terms*—MapReduce, fault tolerance, failure detection**

I. Introduction

Nowadays storage systems face the need of storing large amount of data, created in a rapid period of time. Keeping its consistency, replication, proper authentication and authorization etc., is getting more and more complex. Improper functioning of these properties may have undesirable consequences in different storage systems, especially in those systems whose monitoring should be done in real time, like finance and military domains.

MapReduce represents a programming model for processing and generating large data sets. Its simple philosophy at implementation and usage, automatic parallelism while keeping a powerful scalability, has made huge community interest for MapReduce exploration, especially in environments where data-intensive applications are primary concern. However, on the other side MapReduce performance is not appropriate for HPC (High Performance Computing), although some alternatives have arisen. In-between them is the possibility to improve the performance and the reliability by means of the application of an improved failure detection.

Encouraged by this, my next research steps will be devoted into the possibility of using proper failure detection mechanisms as a decent distributed computing service, in order to improve the fault tolerance of MapReduce framework.

The main research focus during my doctorate period at Universidad Politécnica de Madrid (UPM) is part of the FP7 SCALUS project [1]. This work will be devoted into the possibility of using proper failure detection mechanisms as a decent distributed computing service, in order to improve the fault tolerance of MapReduce framework. All the design and implementation aspects will be concentrated on Hadoop [2], as the most wide-spread implementation of MapReduce.

II. Discussion

A. MapReduce

It's been almost a decade that MapReduce has been commented by researchers, including the database community. Even though its benefits are questioned when compared with parallel databases, some authors suggest that both of the approaches have their own advantages, and do not bring a potential *obsolation* risk for the other opponent [3]. MapReduce advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs [4]. It is because of this, MapReduce is growing popularity and use for different large-scale computing environments, such as in Facebook, Inc., Yahoo! Inc., Microsoft Corporation, etc.

The most common implementation of MapReduce comes as a part of the open-source Hadoop framework [2]. By default, Hadoop uses the Hadoop Distributed File System (HDFS) as the underlying storage backend, but it was designed to work on other file systems as well. A new network striping module for HDFS, created by Facebook, is now available [5].

Every MapReduce execution needs a special node, called master; the other nodes are called workers. While the master keeps several data structures, like the state and the identity of the worker machines, the worker nodes are assigned different tasks by the master.

MapReduce tasks are re-executed in case of failure, and a potential failure of a single master causes an additional bottleneck. Idle nodes with the corresponding replicas have more priority to be selected when you need to re-execute a task which failed in the primary worker. In some cases, migration is needed: the failed task has to migrate to a node that does not hold the necessary data. All these processes waste time, and in fact, because of this MapReduce performance it seems not appropriate for HPC (High Performance Computing). However, it is observed from that the detection of the failed worker tasks in Hadoop have a certain delay, yet not solved, while causing [6]:

- Execution time for a small job increases significantly
- A healthy node to possibly be added into blacklist by mistake
- Too many unnecessary backup tasks scheduled

To conclude, Hadoop couples failure detection and recovery with overload handling into a conservative design with conservative parameter choices, causing a bigger slowdown in reacting to failures and also exhibiting large variations in response time under failure [7].

B. Failure detectors

Asynchronous distributed systems are characterized by the fact there is no bound on the time it takes for a process to execute a computation step, or for a message to go from its sender to its receiver. This is why these systems are usually called "time-free" systems [8].

Apart from their advantages, in asynchronous systems [9], Consensus and NBAC (Non-Blocking atomic commit) are problems that do rise often, needing a solution that is efficient and optimized too. Knowing that Consensus and NBAC (Non-Blocking atomic commit) are unsolvable in asynchronous systems with process crashes (even if communication is reliable), one way to circumvent such impossibility results is through the use of unreliable failure detectors.

Failure detectors are abstract devices that offer information about the operational status of processes in a distributed system. It is believed that the failure detector abstraction is a fundamental one and should sit as a first-class citizen of a distributed programming library. Additionally, failure detectors are important because of the possibility to classify problems in distributed computing [10].

Failure detectors may be divided in perfect or eventual. Perfect detectors may report some process to have crashed, immediately with the first sings of unresponsiveness, while the eventual detectors report a level of suspect. The type of failure detectors can be decided by the level of accomplishing two main properties:

1) Completeness, and
2) Accuracy.

In [11], based on the above mentioned properties and their corresponding variants, authors present a table of different failure detectors, in total eight (8). Basically, every failure detector should belong to one of the groups.

On the other side, there are researchers that have tried applying their failure detectors for a particular problem. From this aspect, an application based taxonomy may be derived as for:

- Consensus/Atomic Broadcast
- Interactive Consistency
- Atomic Commitment
- Register Implementation, etc.

Anyway, there are cases when the failure detection algorithm have been modeled with the possibility of adjusting to different environments; in such a group belongs the φ accrual failure detector [12]. The real significance of the algorithm lies of decoupling the monitoring with the interpretation, in contrary with the other traditional failure detectors. Here basically every application may have its own interpretation when suspecting failures.

A good initial summary of the research in failure detection has been made in [11]. In fact, this work may be considered an unofficial start, where the authors propose to add a failure detector as an external mechanism but that is unreliable and that can do mistakes, while solving Consensus and Atomic Broadcast in asynchronous systems. After this, many other research works succeeded.

Failure detection based on Gossip-based protocol has been discussed in [13]. The authors try to add another feature to the algorithm, while combining it with partial broadcast algorithm, and with this to present a prototype protocol. It is even mentioned that Gossip-based protocol has its weakness, when many failures happen in the system, which as a consequence suggests time-to-time broadcast to be done. In [14], there have been proposed two different failure detection algorithms, that may work with or without stable storage. [15] has its importance because it divides failure detection and membership as two things, additionally not going with heart beating method. There may be noticed the basic approach, when they present the algorithm as two components.

More concrete work of failure detectors in Storage Systems has been proposed in [16]. Here, heart beating has been combined with some intelligent prediction model, based on the previous heart beating information. The good thing is that you may tune the protocol to as many nodes as you want in the system, making subgroups, for example based on different data centers or similar.

Furthermore, there exists works relating failure detection in MapReduce itself [7] [6]. In [7] the problem of today's MapReduce model has been explained well, giving sufficient proofs, but no real solution has been proposed. While in [6], the authors try to implement an algorithm with fine-tuning the failure detection after the estimation of each job as been done. Additionally, it has been added a kind of reputation layer that chooses nodes healthy nodes for executing tasks, depending on the nodes behavior history.

III. Proposal

Analyzing the MapReduce execution task, we may say that the problem resembles to the NBAC problem. This, because every task needs results from each worker, otherwise it cannot give an appropriate output. So, suspecting and detecting each node (master or worker) in a right time is necessary.

Looking into the existing research papers, we see no real challenge in solving the existing fault tolerance issues in MapReduce, particularly the issue related to detecting failures. Being convinced that this is not enough, our idea encompasses these steps:

1) Job Estimation Application based (coarse-grained) - as it was mentioned before, there are cases in which job estimation has been done, but it was fine-grained. As this may lead to an additional performance overhead, we think to tune our algorithm of estimating jobs but from the application based level.
2) Improved collaboration between NameNode/DataNode and JobTracker/TaskTracker - main tools in Hadoop are HDFS and MapReduce. As HDFS consists of NameNode and DataNodes, while MapReduce has JobTrackers and TaskTrackers, we see it necessary that these two layers improve their collaboration, while exchanging the information with each other.
3) Customizable φ accrual FD - it was chosen this algorithms because of its high level of customization possibilities.
4) Applications information interchange - finally, this step belongs to additional benefits, when some applications have longer jobs, while the other applications have shorter lasting jobs, with the possibility of information interchange. This, in order that different applications are prepared faster for job re-execution.

The ongoing schedule should consist in:

- Completing the model implementation - basically, this part should contain the first prototype that validates that our idea is sane and feasible in practice.
- Model validation - short-term future works should consist in the model validation by the end of this school year. Extensive experiments should follow in a non-small testbed environment like Grid5000 [17] experimental testbed.
- Fine-tune the model for other frameworks - MapReduce algorithm is already perfect for parallelization and thus is a good candidate for implementation on architectures with specialized hardware, such as GPUs and FPGAs. We think that our contribution will make easier detecting the failures also in these heterogeneous hardware environments.
- The modified MapReduce on top of a monitoring system for clouds - another possibility that we are looking into is the attachment of the modified MapReduce framework containing our failure detection mechanism on top of monitoring system for clouds. Basically, the idea consists in trying to benefit from the monitoring system information, in order to use it as input to our failure detection mechanism.

Along with the theoretical work should continue experiments, starting with combination of different MapReduce frameworks and failure detection algorithms we have nowadays, to finally reach the level of introducing MapReduce with our own added value algorithm.

References

[1] SCALUS, *SCALUS - Scaling by means of ubiquitous storage*, http://www.scalus.eu/.

[2] T. A. S. Foundation, "Hadoop," http://hadoop.apache.org/, 2011.

[3] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and parallel DBMSs: friends or foes?" *Commun. ACM*, vol. 53, pp. 64–71, January 2010. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629197

[4] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Commun. ACM*, vol. 53, pp. 72–77, January 2010. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629198

[5] N. Rutman, "Map/Reduce on Lustre - Hadoop Performance in HPC Environments," Langstone Road, Havant, Hampshire, PO9 1SA, England, Tech. Rep., 2011. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629197

[6] H. Zhu and H. Chen, "Adaptive failure detection via heartbeat under hadoop," in *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific*, dec. 2011, pp. 231 –238.

[7] F. Dinu and T. S. E. Ng, "Hadoop's overload tolerant design exacerbates failure detection and recovery," in *6th International Workshop on Networking Meets Databases (NetDB'11)*, June 12 2011, pp. 1 – 7.

[8] M. Reynal, "A short introduction to failure detectors for asynchronous distributed systems," *SIGACT News*, vol. 36, pp. 53–70, March 2005. [Online]. Available: http://doi.acm.org/10.1145/1052796.1052806

[9] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The weakest failure detectors to solve certain fundamental problems in distributed computing," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, ser. PODC '04. New York, NY, USA: ACM, 2004, pp. 338–346. [Online]. Available: http://doi.acm.org/10.1145/1011767.1011818

[10] F. C. Freiling, R. Guerraoui, and P. Kuznetsov, "The failure detector abstraction," *ACM Comput. Surv.*, vol. 43, pp. 9:1–9:40, February 2011. [Online]. Available: http://doi.acm.org/10.1145/1883612.1883616

[11] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, pp. 225–267, March 1996. [Online]. Available: http://doi.acm.org/10.1145/226643.226647

[12] X. Dfago, P. Urbn, N. Hayashibara, and T. Katayama, "The accrual failure detector," in *RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology*, 2004, pp. 66–78.

[13] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, ser. Middleware '98. London, UK: Springer-Verlag, 1998, pp. 55–70. [Online]. Available: http://portal.acm.org/citation.cfm?id=1659232.1659238

[14] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," *Distrib. Comput.*, vol. 13, pp. 99–125, April 2000. [Online]. Available: http://portal.acm.org/citation.cfm?id=1035750.1035753

[15] A. Das, I. Gupta, and A. Motivala, "Swim: scalable weakly-consistent infection-style process group membership protocol," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002, pp. 303 – 312.

[16] Y. Wan, Y. Luo, L. Liu, and D. Feng, "A dynamic failure detector for p2p storage system," in *New Trends in Information and Service Science, 2009. NISS '09. International Conference on*, 30 2009-july 2 2009, pp. 15 –19.

[17] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, "Grid'5000: a large scale and highly reconfigurable grid experimental testbed," in *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, nov. 2005, p. 8 pp.