Peer-to-Peer MapReduce Platform

Darhan Akhmed-Zaki[1], Grzegorz Dobrowolski[2] and Bolatzhan Kumalakov[1]
[1]*Mechanics-Mathematics Faculty, al-Farabi Kazakh National University, Al-Farabi ave. 71, Almaty, Kazakhstan*
[2]*Department of Computer Science, AGH University of Science and Technology, Cracow, Poland*

Keywords: Peer-to-Peer Computing, MapReduce Framework, Multi-agent Systems.

Abstract: Publicly available Peer-to-Peer MapReduce (P2P-MapReduce) frameworks suffer lack of practical implementation, which significantly reduces their role in system engineering. Presented research supplies valuable data on working implementation of P2P-MapReduce platform. Resulting novelties include advanced workload distribution function, which integrates mobile devices as execution nodes; novel computing node and multi-agent system architectures.

1 INTRODUCTION

Current research is motivated by a chapter on Peer-to-Peer MapReduce framework presented in (Antonopoulos and Gillam, 2010), where authors use simulation to justify its benefits.

We conduct practical experiment and pursuits two objectives. First of all, theoretical studies presented in (Marozzo et al., 2011) and (Antonopoulos and Gillam, 2010) emphasize that volunteer computing promises considerable increase in system dependability due to self-organization phenomena. That is P2P-MapReduce platform is more likely to finish execution when unexpected node failure accrues or nodes leave infrastructure unpredictably at a run time. We support this argument by conducting computational experiment and presenting derived practical data.

Second, we analyse workload distribution within complex computing infrastructure. System complexity in this case comes from viewing computing architecture as a collection of autonomous devices, which encapsulate control and goal achieving functions (Zambonelli et al., 2003). As a result, devices are not viewed as means of achieving targets, but as active components that solve user defined problems by self-organizing and cooperating.

We bring additional complexity by integrating mobile devices as processing units into the computing infrastructure. As a result, agent accepts *mapper* and *reducer* roles with respect to its subjective self-evaluation. Thus, device capabilities are analyzed and decisions are made dynamically at a run time in a decentralized fashion.

The reminder of the article is organized as follows. Section 2 describes P2P-MapReduce architecture and implementation, while Section 3 defines the workload distribution function that serves as agent decision making tool. Finally, Sections 4 and 5 present computational experiment results and conclusions.

2 MapReduce PLATFORM

In order to implement and test P2P-MapReduce platform we propose architecture, described in Subsection 2.1.

Core of our framework is the high degree of machine autonomy, which is not limited to freedom of accepting or rejecting tasks, but also includes the right to independently change roles from execution to execution; or take numerous roles (reducer and supervisor) at the same time. Agents in this case carry organizational and managerial responsibilities. Computing node intercommunications are governed by agents social interactions, thus, general system architecture fully relies on P2P principles. There are several distinctions from existing architectures.

Unlike in (Gangeshwari et al., 2012), where authors organize multiple data centers (agent supervised) into a hyper cubic structure, we view every machine as an autonomous entity. Hyper cube agents carry supervisory functions for data center infrastructures with pre-installed MapReduce software. Decision making is made on the level of organization; that is accepting or rejecting jobs and optimizing communications. In our approach machines self-organize to

Ahmed-Zaki D., Dobrowolski G. and Kumalakov B..
Peer-to-Peer MapReduce Platform.
DOI: 10.5220/0004330405650570
In *Proceedings of the 5th International Conference on Agents and Artificial Intelligence* (ICAART-2013), pages 565-570
ISBN: 978-989-8565-39-6

actually perform MapReduce jobs without being pre-organized into any structures. Moreover, they do not require installing additional MapReduce software.

In (Marozzo et al., 2011) computing devices (run by agents) are assigned master or slave roles and then master nodes cooperate to organize and manage effective job execution. One master node communicates with the user and organizes task execution, whilst other master nodes monitor it and voluntarily take on control if it fails. Slaves are assigned *map* and *reduce* operations by an active master node, execute code and return the result to the master.

P2P principles, in this case, are implemented to manage master node failure, whilst slave nodes are being controlled. In our approach there is no direct control mechanism, but localized supervision in a form of reducer-mapper and supervisor-reducer relationships. This means that no node has direct control over others, but may indirectly influence execution flow. In such a way we apply the complexity prism and design a system that makes use of agent autonomy in a broader way.

We are also aware of other volunteer MapReduce architectures (Costa et al., 2011) and (Dang et al., 2012), which, however, do not make use of agent-oriented approach.

Reminder of the section describes system architecture and its current implementation in more details.

2.1 Architecture

Our system consists of multiple nodes that interact in order to perform *MapReduce jobs*. Every node may initiate user task, or perform any task offered to it. Process is visualized in Figure 1.

Broker node receives job specification from the user, brakes it into *reduce* and *map* tasks and broadcasts information messages to potential performers (Figure 1). Having received a broadcast message, other nodes evaluate it and issue an offer or do not respond to information message at all. Broker chooses between potential performers on the basis of offer price and readiness to become reducer node, where the lowest offer (or the first lowest offer received, if there is a number of them) wins.

Chosen performer gets confirmation message and looks for supervisors to serve as active backup entities for the execution time. Their primary role is to monitor reducer actions, save peer state and, if unexpected failure accrues, to restart it in the last available state.

When supervision is set, performer requests the job, gets *reducer* status and searches for mappers and leaf reducer nodes by following the same protocol. In such a way nodes self-organize in a tree structure until

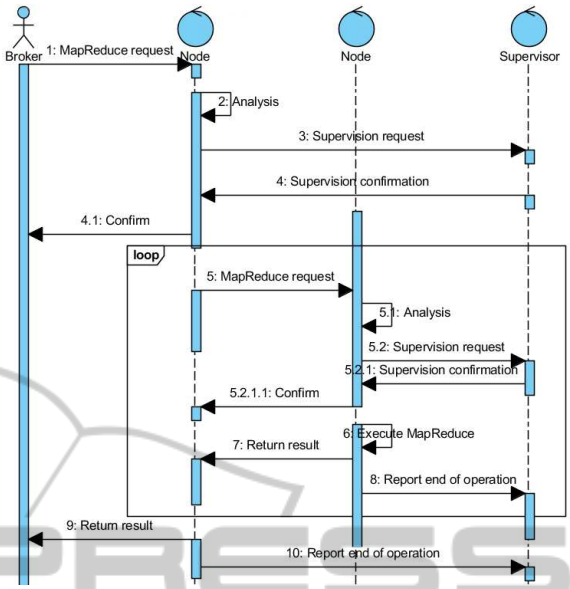

Figure 1: UML Sequence Diagram describes agent interactions when performing a job.

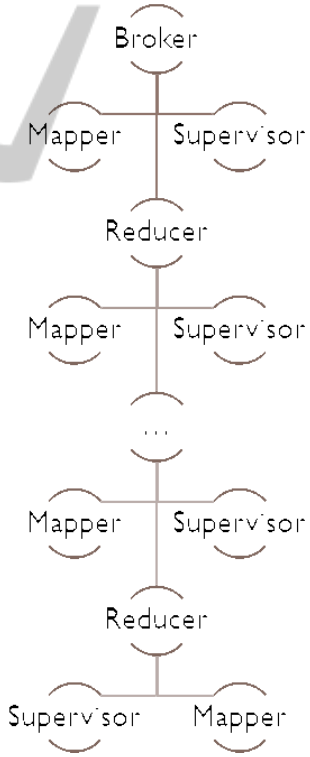

Figure 2: Diagram describes MapReduce tree structure that is formed by peer nodes before reduce operation begins.

the last required node is added to the tree structure (Figure 2).

Reducer nodes serve as a core of the tree structure, whilst mapper nodes do not have leaf nodes at

all. This means that every reducer node tries to find leaf reducer and if none is fount, tree development terminates. Number of mapper nodes per reducer is set by system developer, as well as number of supervisors required per reducer. One supervisor may supervise numerous reducers and take on mapper or reducer roles at the same time depending on self evaluation results.

Job execution terminates when all mapper nodes pass their results to corresponding reducers and all reducers in a hierarchy pass processed results up to the level of the broker node.

It is also remarkable that the broker agent does not have any mapper nodes, but every reducer node does. Supervisors monitor reducer nodes and reducers carry out supervision role for their mapper nodes.

On the node level there are three main components: broker agent, performer agent and compiler/interpreter.

When node receives a broadcast message, it is handled by the performer agent. Performer agent evaluates nodes current state and makes a decision whether to form an offer message or to do nothing. If an offer is issued to the requesting node, performer is responsible for handling upcoming operations. That is finding supervisors, passing executable code to the interpreter/compiler, retrieving execution results and sending them to the destination.

Broker agent is created by interpreter/compiler when user wants to launch a computing task. It is responsible for broadcasting information messages, finding itself supervisor and reducer nodes and handling organizational communication at execution time.

Figure 3 presents class diagram, which visualizes nodes structure, system roles and their relationship to the user interface.

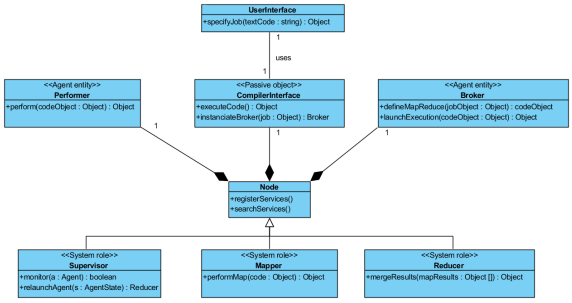

Figure 3: UML Class Diagram describing nodes composite structure, their roles and relationship with user interface.

More details on agent decision making are explained in Section 3.

2.2 Implementation

We implemented system prototype using JADE (Bellifemine et al., 2007) due to cross platform properties and well established development tools of Java. Moreover, availability of standard Jade behaviors allowed convenient grouping of individual operations using ParallelBehaviour and SequentialBehaviour classes, supplied in JADE-4.2.0 distribution.

Executable code (written in Lisp) is encapsulated into ACLMessage object and passed between agents. Code is executed on Java Virtual Machine using Clojure-1.4 (PC and server machines) and JScheme-7.2 (mobile devices). It is important that standard JScheme libraries had to be modified in order to work with floating point numbers and files on mobile devices. Apart from that software was used as supplied in standard distribution.

Agent initialization includes publishing two advertisements: first, supervision services, second, MapReduce services. As noted before there is no predefined mapper or reducer role, because it depends on self-evaluation at execution time.

Supervisor does not copy reducer state directly, but knows about changes by listening to duplicated messages, sent to the reducer. In other words it updates state record when receives mapper and leaf reducer message duplicates.

In order to describe job submission and failure recovery mechanisms we take example scenario that corresponds to the algorithm described in Figure 1:

1. All agents register their supervision and execution services using DFAgentDescription class;
2. When job arrives broker agent breaks it down into map and reduce operations and launches initiateExecution behavior, which is an extention of the Jade Behaviour;
3. When perspective performer receives offer it decides to execute the reduce task or not. If decision is positive, agent instantiates SequentialBehaviour object with unique CoversationId;
4. Broker tracks best offer among received within specified timeout and sends confirmation;
5. Perspective performer becomes reducer node and searches for supervisors by broadcasting ACLMessage.INFORM. When answers are received first three answer owners become supervisors and their addresses are put into offerMessage. Then offerMessage is broadcasted to new potential performers;
6. Supervisors monitor their reduce node. If a call timeout is reached, supervisor tries to re-launch

the agent. If host is unreachable, supervisor tries to assign the task to other host at last reducer state;

7. When reducer receives mapper and leaf reducer results it uses provided reduce code and data by passing it to the Clojure compiler or scheme interpreter for execution. Result is encapsulated and returned to the specified destination;
8. When reducer part is done it notifies supervisors and all of them delete their job SequentialBehaviour object.

3 WORKLOAD DISTRIBUTION FUNCTION

Task of the distribution function is to distribute workload between computing nodes as even as possible. Formally it may be specified as follows.

Let us denote executable task by J and its step by k, such as $J = \{k_1, k_2, \ldots, k_n\}$, where all steps are performed by a set of computing nodes $A = \{a_1, a_2, \ldots, a_m\}$. k in this case is an uninterrupted process which is performed according to its specification. If step k_n may be performed by node a_m, we denote it as a mapping function $k_n \to a_m$.

Workload distribution means that mappings between different nodes in A should be distributed as even, as possible. Let us use *price* as a derivative of available resources, workload and other parameters, which reflects comparative workload of individual device. As a result, every successful mapping $k_i \to a_j (1 \leq i \leq n, 1 \leq j \leq m)$ gets computing price p_{ijk} assigned by an accepting computing node. Following is the price function:

$$p_{ijk} = f(\omega_k, p_b, b_l) \tag{1}$$

Here, p_b denotes basic resource price, which is set by device owner; b_l denotes battery load; and ω_k denotes resources availability at the time, when step k arrives.ω_k has following descrete values:

$$\omega_k = \begin{cases} 1 & \text{device free, can map and reduce} \\ 0.6 & \text{device buisy, can map and reduce} \\ 0.3 & \text{device can map only} \\ 0 & \text{otherwise} \end{cases}$$

Computed price for different mappings may not be the same $p_{ijk} \neq p_{ljk}$, where $i \neq l$ and $1 \leq i, l \leq m$. If they are equal, the conflict is resolved on the *first come first served* basis.

After price is computed tuple $((\rho(\omega_{ijk}), p_{ijk})$ is returned to initiator node, where p_{ijk} is computer price and $\rho(\omega_{ijk})$ is determined as follows:

$$\rho(\omega_{ijk}) = \begin{cases} 1 & \omega_{ijk} > 0, \text{ want to supply services} \\ 0 & \text{otherwise} \end{cases}$$

Then, issuer returns result of function $\varphi(p_{ijk})$, which determines executor node.

$$\varphi(p_{ijk}) = \begin{cases} 1 & \rho(\omega_{ijk}) = 1,\ p_{ijk} \text{ of } k \to a_i \text{ lowest} \\ 0 & \text{otherwise} \end{cases}$$

We, also, include client balance *cb* value in order to represent the amount of money user can spend on services.

Using values stated above distribution function is formulated as follows:

$$\min_p \sum_{i=1}^{n} \rho(\omega_{ijk}) \varphi(p_{ijk}) \tag{2}$$

Subject to:

$$\sum_{i=1}^{n} \rho(\omega_{ijk}) \varphi(p_{ijk}) \leq cb \tag{3}$$

$$\sum_{i=1}^{n} \rho(\omega_{ijk}) \varphi(p_{ijk}) > 0 \tag{4}$$

Objective function (2) minimizes overall cost of performing MapReduce job by choosing lowest price at each step. Constraints ensure that overall solution cost is always lower than client balance (3) and at least one path of job execution exists (4).

Objective function is implemented as an aim of every agent to choose cheapest offer available. In its turn, offer is a derivative of unused physical resources of the host device. In such a way it is ensured that next step performer is the one, who has bigger proportion of free resources.

4 EXPERIMENT AND EVALUATION

4.1 Experiment Design

In order to test our P2P platform against master-slave based systems we have set up an infrastructure that consists of 23 PCs with Intel Core i3 processors and 2 Gb RAM; 2 HP servers with 4 Core Intel Xeon Processors and 6 Gb RAM. To fully test the P2P infrastructure we also included 12 mobile devices run under Windows Phone and Android operation systems into P2P-MapReduce runs.

All nodes (except mobile devices) got master-slave based Hadoop-1.0.4 Stable release and newly designed multi-agent MapReduce system installed on

them. Each system was tested by performing three rounds of 100 runs of the same MapReduce task one after another. When Hadoop runs were performed P2P-MapReduce system was off and visa versa.

At the first stage probability of the node (including head node) accepting the task and crashing was programmed to be 5%, at second stage 10% and third 20%. Crashing in this case means that every time node receives the task it calls a method, which with the given probability calls finalize(). Frequency of the task being launched is controlled by randomizer that generates pauses between launches in the range between 0 and 2 seconds.

To keep record of *finished jobs* and *workload distribution* performer agents sent acknowledgement messages and CPU load data to the Logger agent. Collected data was plotted to graphs using MS Excel software.

4.2 Experiment Results

Figure 4 presents percentage of finished jobs given 100 runs of the MapReduce task. From the results it is seen that developed P2P platform shows better dependability at the defined probabilities of failure. This corresponds to the theoretical result of (Antonopoulos and Gillam, 2010). However, our implementation of P2P-MapReduce platform has shown 99% of all jobs finished at 5% failure rate, which further decreased down to 82% with failure probability at 20%, unlike predicted 100%. Preliminary analysis of failure causes indicate that it might be due to supervisors trying to restart reducer agents at overloaded devices. Further analysis should give clarification on this issue.

This result satisfies the first aim of the research and supports hypothesis on the P2PMapReduce platform dependability.

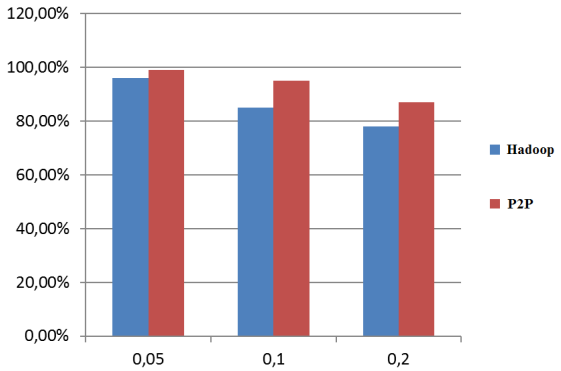

Figure 4: Competitive analysis of P2P platform and Hierarchical platform performance given different node failure probabilities.

In terms of second aim: workload distribution, platform shows performance presented in Figure 5. It is seen that workload is evenly distributed between computing nodes, except the highest workload is recorded at devices number 13, 19 and 25. This is due to the fact that those are mobile phones and map operation takes considerably more resources to be computed.

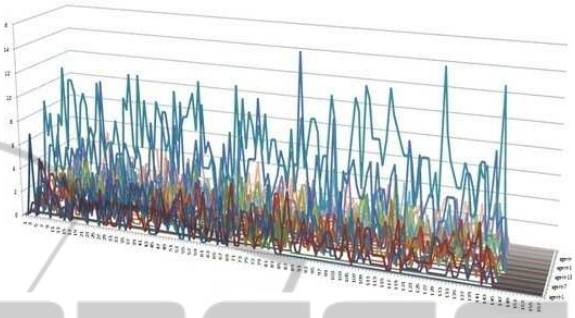

Figure 5: Workload distribution in P2P computing environment (x axis - CPU load; y axis - CPU hours; z axis - agent identifier).

In order to make deeper analysis let us consider average workload on device types: server machines, mobile devices and personal computers (Figure 6). It confirms higher load on mobile devices, whilst PCs and server machines use little amount of their resources.

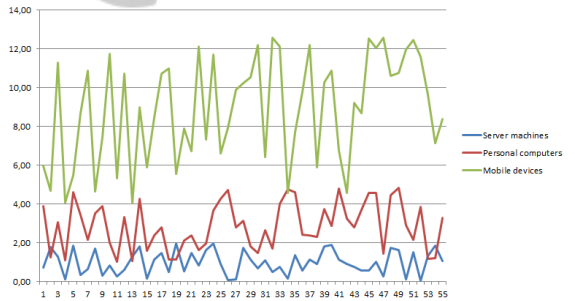

Figure 6: Average workload distribution over device types in P2P computing environment.

This result confirms that developed system sustains desired behavior under system complexity. Thus, there is a perspective in applying alike system for infrastructures where computing nodes are not guaranteed to be available for a long time.

5 CONCLUSIONS

Presented research provides some practical insides to the P2P-MapReduce computing concept. In particular they confirm that system maintains desired workload balancing behavior in a complex environment

and is able to self-organize and self-reorganize dynamically without being explicitly programmed to do so.

On the other hand, system performance has not been studied yet, whist it is one of the most important factors when choosing a computing platform. Thus, there is a need to analyze execution efficiency and compare it to avaliable P2P-MapReduce platform evaluations.

Further research is going to concentrate on the execution performance. In particular we shell design or adopt an agent learning framework that is to manage system efficiency by affecting agents social behavior.

Finally, it is worth pointing out considerable limitations of the presented research. First, implemented P2P platform is simple and perspective platform development may lead to changes in dependability in any way. Second, experiment installation of Hadoop might not be optimal and results may be misleading to some extent.

REFERENCES

Antonopoulos, N. and Gillam, L. (2010). *Cloud computing: Principles, Systems and Applications*, volume 0 of *Computer Communications and Networks*, chapter 7, pages 113–126. Springer-Verlag London Limited, London, UK.

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, NJ.

Costa, F., Silva, L., and Dahlin, M. (2011). Volunteer cloud computing: Mapreduce over the internet. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1855–1862. IEEE Computer Society.

Dang, H. T., Tran, H. M., Vu, P. N., and Nguyen, A. T. (2012). Applying mapreduce framework to peer-to-peer computing applications. In Nguyen, N. T., Hoang, K., and Jedrzejowicz, P., editors, *ICCCI (2)*, volume 7654 of *Lecture Notes in Computer Science*, pages 69–78. Springer.

Gangeshwari, R., Janani, S., Malathy, K., and Miriam, D. D. H. (2012). Hpcloud: A novel fault tolerant architectural model for hierarchical mapreduce. In *ICRTIT 2012*, pages 179–184. IEEE Computer Society.

Marozzo, F., Talia, D., and Trunfio, P. (2011). A framework for managing mapreduce applications in dynamic distributed environments. In Cotronis, Y., Danelutto, M., and Papadopoulos, G. A., editors, *PDP*, pages 149–158. IEEE Computer Society.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol*, 12(3):317–370.