Research Article Open Access

M.N. Satymbekov*, I.T. Pak, L. Naizabayeva, and Ch.A. Nurzhanov

Multi-agent grid system Agent-GRID with dynamic load balancing of cluster nodes

https://doi.org/10.1515/eng-2017-0054
Received June 9, 2017; accepted August 24, 2017

Abstract: In this study the work presents the system designed for automated load balancing of the contributor by analysing the load of compute nodes and the subsequent migration of virtual machines from loaded nodes to less loaded ones. This system increases the performance of cluster nodes and helps in the timely processing of data. A grid system balances the work of cluster nodes the relevance of the system is the award of multi-agent balancing for the solution of such problems.

Keywords: dynamic load balancing, jade, hadoop-Map reduce, grid systems, multi-agent systems

1 Introduction

Current research is motivated by a chapter on Hadoop-MapReduce framework presented by Google used for parallel computations over very large, several petabytes, data sets in computer clusters. In this work, the work led to conduct practical experiment and pursuits two objectives. First of all theoretical studies emphasize, that volunteer computing promises considerable increase in the system dependability due to self-organization phenomena. That is Hadoop-MapReduce platform, which more likely exists to finish execution when unexpected node failure accrues or nodes leave infrastructure unpredictably at a run time. Furthermore, the study supports the argument by conducting computational experiment and presenting derived practical data. Secondly, in this work analyses workload distribution within complex computing infrastructure were performed. The system complexity in this case comes from viewing computing architecture as a collection of autonomous devices, which encapsulate control and goal achieving functions [1]. As a result, devices are not viewed as means of achieving targets, but as active components that solve users defined problems by self-organizing and cooperating. In addition, the system brings 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 analysed and decisions are made dynamically at a run time in a decentralized fashion. The structure of the presented article is organized as follows: Section 2 describes Hadoop-MapReduce architecture and implementation, Section 3 defines the workload distribution function that serves as agent decision making tool, while section 4 shows multi-agent dynamic cluster load balancing. Finally, Sections 5 and 6 present computational experiment results and conclusions to the work carried out.

2 Hadoop-Mapreduce platform

In order to implement and test Hadoop-MapReduce platform in the study it is proposed the architecture, described in Subsection 2.1. The core of the 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 Hadoop-MapReduce principles. There are several distinctions from existing architectures. Unlike in the work presented by Gangeshwari et al. [2], where study shows multiple data centres (agent supervised) into a hyper cu-

*Corresponding Author: M.N. Satymbekov: Al-Farabi Kazakh National University, Institute of Information and Computational Technologies, Almaty, Kazakhstan, E-mail: m_satymbekov@mail.ru
I.T. Pak: Al-Farabi Kazakh National University, Institute of Information and Computational Technologies, Almaty, Kazakhstan, E-mail: pak.it@mail.ru
L. Naizabayeva: Al-Farabi Kazakh National University, Institute of Information and Computational Technologies, Almaty, Kazakhstan, E-mail: naizabayeva@gmail.com
Ch.A. Nurzhanov: Al-Farabi Kazakh National University, Institute of Information and Computational Technologies, Almaty, Kazakhstan, E-mail: darkeremite@ya.ru

bic structure, we view every machine as an autonomous entity. Hyper cube agents carry supervisory functions for data centre 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 the current approach machines self-organize actually perform MapReduce jobs without being pre-organized into any structures. Moreover, the machines do not require installing additional MapReduce software. In the work [3] 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. Hadoop-MapReduce 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. In the study the attention has been paid to the other volunteer MapReduce architectures [4] and [5], 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

The presented system in this work 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 Fig. 1. Broker node receives job specification from the user, brakes it into reduce and map tasks and broad-casts information messages to potential performers (Fig. 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 obtains confirmation message and searching 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 the reducer nodes by following the same protocol. In such a way nodes self-organize in a tree structure until the last required node is added to the tree structure (Fig. 2).

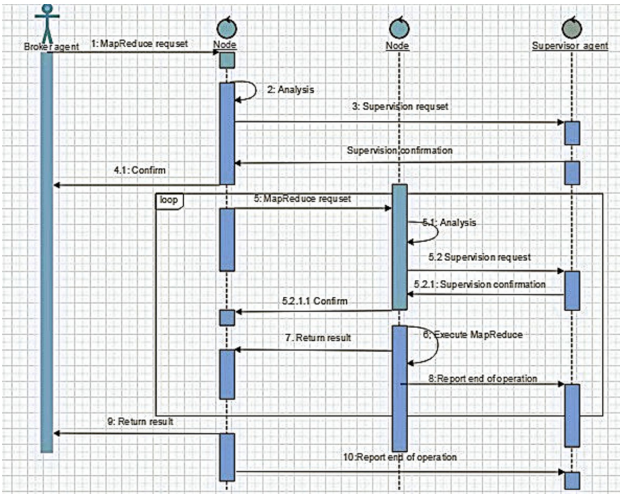

Figure 1: UML Sequence Diagram describes agent interactions when performing a job.

The 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. The broker agent is created by interpreter/compiler when user

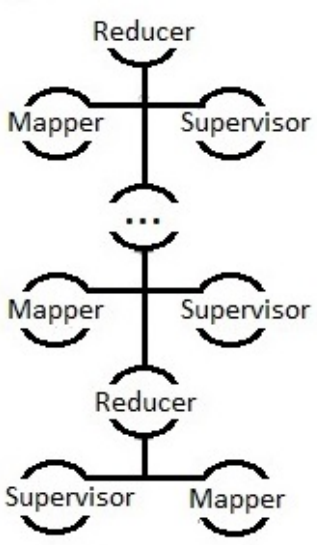

Figure 2: Diagram describes MapReduce tree structure that is formed by peer nodes before reduce operation begins.

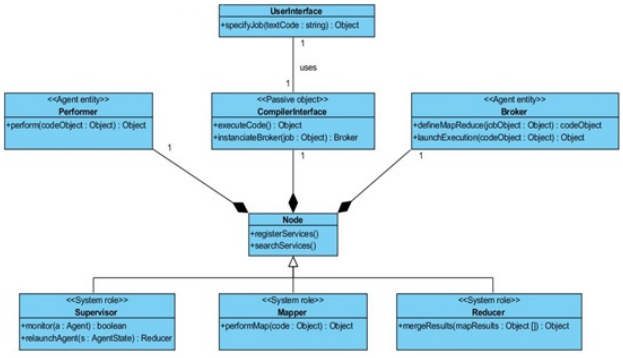

Figure 3: UML Class Diagram describing nodes com- posite structure, their roles and relationship with user interface.

wants to launch a computing task. It is responsible for broadcasting information messages, finding itself supervisor and the reducer nodes and handling organizational communication at execution time. Fig. 3 presents class diagram, which visualizes nodes structure, system roles and their relationship to the user interface.

2.2 Implementation

The study implemented system prototype using JADE [6] due to cross platform properties and well established development Java tools. Moreover, availability of standard Jade behaviours allowed convenient grouping of individual operations using parallel behaviour and Sequential Behaviour classes, supplied in JADE-4.2.0 distribution. Executable code (written in Lisp) is encapsulated into ACL message object and passed between the agents. The code is executed on Java Virtual Machine using Clojure-1.4 (PC and server machines). The agent initialization includes publishing two advertisements: first, supervision services, second, MapReduce services. As noted before there is no pre-defined 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 the example scenario that corresponds to the algorithm is described in Fig. 4.

1. All agents register their supervision and execution services using DFAgent description class;
2. When job arrives the broker agent breaks it down into map and reduce operations and launches initiateExecution behaviour, which is an extension of the Jade Behaviour;
3. When perspective performer receives offer it decides to execute the reduce task or not. If decision is positive, the agent instantiates Sequential Behaviour 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 ACL Message INFORM. When answers are received first three answer owners become supervisors and their addresses are put into offer Message. Then offer Message is broadcasted to a 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;

3 Workload Distribution Function

The task of distribution function is carried out 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 $A = \{a_1, a_2, a_3, \ldots, a_n\}$. k in this case is an uninterrupted process which is performed according to its specification. If step k_n may be performed by node am, denoted here 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 $k_i \to a_j (1 \le i \le n, 1 \le j \le m)$ 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:

$$\omega_k \begin{cases} 1 & \text{device free, can map and reduce} \\ 0.6 & \text{device busy, can map and reduce} \\ 0.3 & \text{device can map only} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Computed price for different mappings may not be the same $p_{ijk} \neq p_{ijk}$ where $i \neq l$ and $1 \le i, l \le m$. If they are equal, the conflict is resolved on the first come first served basis. $((p(\omega_{ijk})), p_{ijk})$ is return to initiator node, where p_{ijk} is computer price and $p(_{ijk})$ is determined as follows:

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

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

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

also, includes 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:

$$\min_p \sum_{n=1}^{n} p(\omega_{ijk})\varphi(p_{ijk}) \tag{3}$$

Subject to:

$$\sum_{n=1}^{n} p(\omega_{ijk})\varphi(p_{ijk}) \le cb \tag{4}$$

$$\sum_{n=1}^{n} p(\omega_{ijk})\varphi(p_{ijk}) > 0 \tag{5}$$

The objective function (2) minimizes overall cost of performing MapReduce job by choosing the 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). The 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 Dynamic cluster load balancing

The above distribution function solves the problem of distributing data to compute nodes. Computing nodes can have different characteristics that will lead to the load of cluster nodes. This section is aimed at dynamic load balancing between cluster nodes. The problem of balancing the computational load of a distributed application arises because:

1. the structure of the distributed application is heterogeneous, different logical processes require different computational capacities;
2. The structure of a computational complex (for example, a cluster) is also heterogeneous, i.e. Different computing nodes have different performance;;
3. the interstitial interaction structure is not homogeneous; The communication lines connecting nodes can have different characteristics of throughput.

4.1 Multi-agent dynamic load balancing system

The dynamic balancing system is multi-agent; it consists of a set of agents of different types:

- Agent-sensor of the computing node;
- The sensor agent of the simulation model;
- The agent of analysis;
- The migration agent;
- Distribution agent

Agents of each type act according to their scenario to achieve the goal, and together they realize the balancing of the distributed simulation model. Agents are hardware and software entities, they can act autonomously. Agents interact with each other and with the external environment.

Agents can be defined as follows: $Agent =< S, Eff, Prog., I, O, R, M, R, E, G, B >$ where,

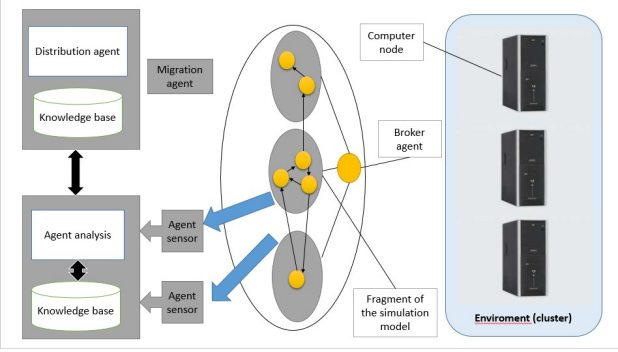

Figure 4: The architecture of the multi-agent balanc- ing system.

1. Sense are the agent's sensors, the functions by which the agent receives information about the external environment.
2. eff - effectors, functions by which the agent acts on the external environment.
3. Prog: donates PO donates transformation of the input information (I) into the output (O).
4. MR donates meta-correction,
5. R donates the rules by which the agent acts (meta-rules and rules Are characteristic for cognitive agents).
6. E donates the external environment (computing environment: network, multiprocessor computer, cluster, GRID).
7. G donates the target that the agent is trying to reach.
8. B donates broker agent.

The agents of analysis and distribution are defined as cognitive. The agents sensors and migration agent are reactive. In Fig. 1 the architecture of the balancing system at each node is presented. The user, based on the knowledge of the model (knowing how the model should work), modifies the balancing rules. Based on these rules, the agents will decide to move the model objects from one computing node to the another. At each compute node, there are five different types of agents. Let us consider in detail the significance of each of them. The agent-sensor of the computer system collects the required data about the state of the computing node. Among these data: the load on the compute nodes, the load of the communication lines. When collecting information, use performance counters. The sensor agent of the simulation model monitors the change in the state of the imitation model objects located on the node. As data that the sensor agent delivers as the output information the frequency of the event, the frequency of receiving and sending messages from the poles, the frequency of the state change. The analysis agent interrogates the analysis agents at certain intervals, decides (using the rules), whether there is a need for balancing. If this is the case, then the distribution agent is contacted. The distribution agent is the source of "knowledge" about the surrounding environment (neighbour objects located on neighbouring computational nodes, statistical information about neighbouring nodes) for the analysis agent. This knowledge is intended to clarify the rules, guided by which the agent decides on the need for balancing. The broker agent controls with all agents and controls the operation of nodes, when the node completed its part of the work. The broker agent takes part of the task from the other node and mixes it to a free node. Thus, the entire node always is in process. The rule-based distribution agent (the set R) decides which model objects need to transfer, and selects the target network nodes.

The distribution agent, by selecting the objects for a transfer to other computing nodes, refers to the neighbouring distribution agents. Next, the distribution agents are synchronized, as a result of which the agent-leader is determined (it is assumed that at the beginning of the synchronization algorithm execution, all processes are in the same state - the distribution agents are ready to stop the system). After the synchronization algorithm is executed, the lead agent stops the simulation process by sending a corresponding message to the simulation system. The agents then request the necessary objects from the system (in the form of streams of serialized data). The migration agent moves objects to other nodes. The migration agent passes the objects of the simulation model to the agents of distribution of the target nodes. After the migration is complete, the distribution agents are synchronized again and the simulation system is started. The modelling process continues. At the same time, the monitoring of the state of computing nodes and the redistribution of the load continue.

5 Experimental results

The system complexity in this case comes from the notion of self-organization and computing resources autonomy in the domain of high-performance computing. The notion of complex software engineering is a well-established area of research that traces back to adaptive systems and artificial intelligence. Its core is built around an idea that software systems capable of adapting to the changing environment without direct commands of an external actor. In terms of practical advantage, this would mean self-managing system that needs little maintenance effort and provides high system dependability due to its design in return. However,

there is little evidence of such systems being exploited on industrial scale. This may happened due to high theorization of the matter and little work on the engineering methods for complex software systems. For instance, despite theoretically proving all the benefits of such systems there is very little evidence of its empirical, working characteristics. In such a case, any justification of complex system design can be trickled down to with no evidence of the basic practical research and argued against. Given this situation, it is highly relevant to study such phenomena empirically and to present some clarification of real system behaviour, to investigate the effects of the proposed load balancing system, a multi-agent system will be implemented to provide a testing platform of the proposed load-balancing model.

As a problem, the Mean-Shift algorithm was implemented to segment and repair damaged video, the data was taken from the Al-Farabi University in Almaty. The whole system was written in Java using JDK 1.6 and was implemented on a cluster of machines, running Linux 2.6.9 (Scientific Linux 4.5). The configurations of these 30 machines with the following characteristics: processor Intel core i3 RAM 3 GB, Intel core i5 RAM 4 GB, Intel core i7 RAM 8 GB.

These machines reside at the Analysis and Processing of Big Data Laboratory, at the Institute of Information and Computational Technologies in Almaty. All the machines possesses separate hard drives, but the same set of files are shared on every one of them. This is to ensure that all the related initialization and configuration files are accessible from every machine. One host is the central host that holds the Main-Container, this host will invoke the whole system start up on each host. As a result of laboratory testing, we took measurements of the root-mean-square deviation of the load of 30 computational nodes, between which virtual machines were balanced. The deviation limit is set at 7%. The results of taking before and after balancing are shown in Fig. 5.

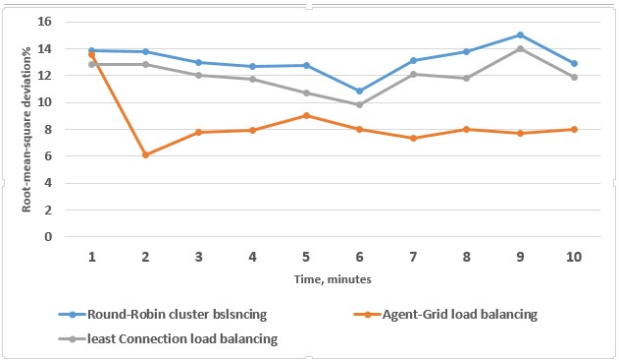

Figure 5: Root-mean-square deviation of node load.

6 Conclusion

The presented research provides some practical insides to the Hadoop-MapReduce computing concept. In particular it was confirmed that the system maintains desired workload balance behaviour in a complex environment and is able to self-organize and self-reorganize dynamically without being explicitly programmed. On the other hand, the system performance was not studied yet, whist it is one of the most important factors when choosing a computing platform. Thus, there is a need to analyse execution efficiency and compare it to available Hadoop-MapReduce platform evaluations. Further research is going to concentrate on the execution performance. In particular, the authors will concentrate on shell design or adopt an agent-learning framework that is table to manage the system efficiency by affecting agents social behaviour. Finally, it is worth pointing out considerable limitations of the presented research. Firstly, implemented Hadoop platform is simple and perspective platform development may lead to changes in dependability in any way. Secondly, experiment installation of Hadoop might not be optimal and results may be misleading to some extent.

References

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

[2] Gangeshwari, R., Janani, S., Malathy, K., and Miriam, D. D. H. (2012). Hpcloud: A novel fault tolerant archi-tectural model for hierarchical mapreduce. In ICRTIT 2012, IEEE Computer Society, pages 179 - 184.

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

[4] Costa, F., Silva, L., and Dahlin, M. (2011). Volunteer cloud computing: Mapreduce over the internet. In IEEE In-ternational Parallel and Distributed Processing Sym posium, pages 1855 - 1862. IEEE Computer Society.

[5] 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.

[6] Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). Developing Multi-Agent Systems with JADE. John Wi-ley & Sons, NJ.