Computer Science • 15 (1) 2014 http://dx.doi.org/10.7494/csci.2014.15.1.35

Włodzimierz Funika
Kamil Mazurek
Wojciech Kruczkowski

AN AGENT-BASED HIERARCHICAL APPROACH FOR EXECUTING BAG-OF-TASKS IN CLOUDS

Abstract *Unlike message-passing applications, "bag-of-tasks" applications (BoTs), whose tasks are unrelated and independent (no inter-task communication), can be highly parallelized and executed in any acceptable order. A common practice when executing bag-of-tasks applications (BoT) is to exploit the master-slave topology. Cloud environments offer some features that facilitate the execution of BoT applications. One of the approaches to control-cloud resources is to use agents that are flexible actors in a dynamic environment. Given these assumptions, we have designed a combination of approaches which can be classified as distributed, hierarchical solutions to the issue of scalable execution of bag-of-tasks. The concept of our system relates to a project that is focused on processing huge quantities of incoming data from a network of sensors through the Internet. Our aim is to create a mechanism for processing such data as a system that executes jobs while exploiting load balancing for cloud resources which use applications such as Eucalyptus. The idea is to create a hybrid architecture which takes advantage of some centralized parts of the system and full distributedness in other parts. On the other hand, we balance dependencies between system components using a hierarchical master-slave structure.*

Keywords cloud computing, bag-of-tasks, load balancing, master-slave paradigm, scalability

1. Introduction

Since the implementation of Amdahl's law in the 1960's, parallel computing has changed. However, the goal has always remained the same: to execute computations as fast as possible. Many models of parallel computing have been introduced over the years. A good example is Grid Computing; computation on multi processor machines and computing on clusters.

Recently, a new approach known as Cloud Computing has been introduced, and it has been gaining more and more attention in business and research. One of the key problems of this type of computation is meeting the needs of customers in regards to the provisioning of sufficient resources, thus ensuring the computation will be performed in accordance to the customers' requirements (for example, setup time constraints). In the practice of cloud computing, a balanced allocation of available computing resources is implied along with the provisioning of additional resources; e.g., virtual machines (whenever those available run out).

A Cloud Computing system [30, 31] must combine dynamic allocation with effective external and internal cloud communication and data storage. Such a system can be used in multi-category computations, but assembling it is not a trivial task. One of the main cloud features is transparency – people who are using a cloud system do not need to know the system architecture, where the machines are placed, and what roles they play. With transparency, the following problem occurs: finding a way to balance the load of machines and utilizing them with optimal performance.

The topic of our research relates to the domain of processing the huge quantities of data coming in from the Internet; e.g., from a network of sensors [14]. Our aim is to create a mechanism for processing this data in the form of a system which executes jobs while exploiting load-balancing procedures for cloud resources. The idea is to create a hybrid architecture in which some parts of the system are centralized and others not, and where we balance the dependencies between system units using a hierarchical structure [13] consisting of masters and slaves. During the research under discussion, our objectives were to investigate the architectural and scalability issues of such an approach and to make its implementation work in a reliable fashion. Some significant reasons that motivated this research are provided later in the paper. In our effort, we have referenced some solutions from past research while attempting to enhance them with our own approaches related to new trends in the domain of computing, data retrieval and storage (such as clouds, huge data processing, and storage mechanisms).

This paper is organized as follows: Sections 2 and 3 focus on cloud computing and requirements for the processing of bags-of-tasks. Section 4 presents the concept of our system. Section 5 describes the system, contains an analysis of some potential issues, provides solutions, and highlights the design decisions we made. In Section 5.1, we concentrate on the system design followed by an example scenario. Sections 5.3, 5.4, and 5.5 present the architecture and system parts, and describe the approaches used. In Section 6, we concentrate on scaling, load balancing, and performance issues. Section 7 contains more-detailed information about the system parts mentioned in

Section 5. Section 8 is focused on tests and some problems that occurred when testing the system on cloud resources, the results of running the example jobs, and benchmark results, followed by conclusions and plans for future work.

2. Research background

Cloud Computing has many definitions. Jeremy Galeman even says, that it is "the phenomenon that currently has as many definitions as there are squares on a chessboard" [20]. In the book "Cloud computing. Principles and Paradigms" [34] by Buyya et al., we find: "Cloud is a parallel and distributed computing system consisting of a collection of inter- connected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements (SLA) established through negotiation between the service provider and consumers".

Another researcher [24] claims that Cloud Computing is a style of performing calculations in which the highly scalable IT infrastructure is accessible to external clients as a service. Providers believe that Cloud Computing is going to revolutionize the IT market and will be widely used in next five to ten years [33]. Many worldwide IT companies such as Microsoft, Amazon, and Google are interested in using clouds in business – they offer computing power, data storage, and services. Clients who decide to use cloud resources may pay less than if they choose On-Premises Servers or Hosted Servers. They will pay only for the usage – not for their own machines, engineers, software, nor bandwidth they do not use. Cloud Computing allows clients to focus on functionality – not infrastructure. Individual customers who decide to use clouds may save money instead of buying faster computers and necessary software. The National Institute of Standards and Technology of US DOC defines Cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential **characteristics**, three **service models**, and four **deployment models**." [27], with those five characteristics being:

- On-demand self-service.
- Broad network access.
- Resource pooling.
- Rapid elasticity.
- Measured service.

There are a number of investigations into the scaling of computations on the cloud. The research under discussion can be classified as: a distributed, hierarchical approach to the problem of the scalable execution of bag-of-tasks, being related to the problem of bag-of-tasks scheduling.

Sets of numerous unrelated, independent (no inter-task communication) tasks are often called bag-of-tasks application (BoTs) [22]. In comparison with the message passing model, they can be highly parallelized and executed in any acceptable order. Such tasks are adopted in multiple scientific research: computational biology, image processing, and massive search engines [28]. The scheduling of independent tasks has been the subject of research over the years [12, 16], and it was shown that the scheduling of them on a set of heterogeneous computing resources is a NP-complete problem. Thus, the research is mainly focused on finding heuristics [22, 23] which will provide a satisfactory solution. A related problem is predicting the completion time of the task (which would be very helpful in scheduling). Both centralized and distributed scheduling of bag-of-tasks approaches are subjects of interest for research [15]. The centralized approach can be easier, faster, safer, and more stable than the distributed one, but it may also be insufficient for large-scale computations. Quite often, it is important to make nodes cooperate. Simple, non-cooperative approaches in which a node is only optimizing its load do not balance the load of the whole system. A common framework is to execute a bag-of-tasks application in the master-slave topology [32], which is proven to be efficient in distributed computations (i.e. a master-slave architecture is adopted in MapReduce implementations). Meta-schedulers can optimize computational workload by combining multiple job schedulers into one system entry and by dispatching tasks among various resources. The expected completion time of a user application is estimated based on the estimates of all running and waiting applications, but such estimations usually turn out to be inaccurate. When tasks are dispatched between multiple clusters, this inaccuracy grows, since rescheduling is performed independently by each resource provider. This time difference is called *stretch factor* [28]. The inaccuracy of predictions can be reduced by such methods as analyzing the scheduling traces (application independent, but problems with heterogeneous workloads) or application profiling (generating run-time predictions, but requires source-code access). We believe that an important feature of a distributed, self-balancing system is use of a standardized method of estimating the execution time.

The scheduling heuristics consist of two phases: *task ordering* and *task mapping*. As for the ordering types, we can distinguish between the unordered one and the one that is based on the sizes of tasks – from small to large or inversely. The task mapping policies include random choice and choice based on the minimum or maximum time already consumed or expected. According to the above, the scheduling of BoT execution can be organized as *FCFS* (First Come First Served), *greedy* (e.g., w.r.t. response), or min-max/max-min (e.g., w.r.t. completion time).

While some computing platforms (e.g., Condor) use a best-effort basis-sharing model without performance guarantees, they are commonly free of charge. Amazon EC2 [1] and other commercial cloud resource providers offer resources with various extents of QoS that can meet user requirements. Allocated machines incur charges only for their usage typically on a per-hour basis. Although different computing paradigms offer the ability to choose proper machines (due to QoS reasons), it is still

a problem to choose machines which hold the best value for the money. A budget-constrained scheduler could be a solution to this problem [29].

Cloud environments offer some features that can be helpful in executing the BoTs:

- Distributed environment.
- Dynamic, scalable resources.
- Quality of Service.
- Virtualised resources.
- Pay-per-use.

In the research on the execution of BoT-type applications, multiple autonomous parties such as consumers, brokers and service providers interact to achieve their individual goals. Agents that are, by default, autonomous can act flexibly in a dynamic environment and, therefore, can be used to solve the issues of interactions, negotiation, cooperation [21], and controlling huge numbers of machines running multiple processes (e.g. synchronization problems). Such agents can be optimized to adapt well to a changing distributed environment and non-constant user demands [11], typical of the bag-of-tasks execution problem.

In the general case, the common framework is executing a bag-of-tasks application in the master-slave topology [32], while the scalability of hierarchical platforms is still a subject of study [25, 26, 18, 35]. BoTs scheduling can successfully be performed using agents running in Cloud resources [17, 21, 22]. Combining the above-mentioned approaches leads to a design of a hierarchical, agent-based system running on a set of heterogeneous computing resources and using cloud resources, which is generally close to the assumptions of the research under discussion in this paper.

3. Motivation for research and requirements

Through the analysis of the existing approaches to BoT processing and their implementations (systems), we stated that:

- the systems are often difficult to run,
- they contain errors that are hard to fix,
- their sources are not always available,
- it is worthy of investigation which technologies and solutions are suitable for building a new system,
- creating a system from scratch allows us to tune the system to the needs of investigating the particular aspects we focus on,
- we do not want to enforce any convention which would prevent us from using particular technologies – merely to achieve an independence of the conceptual and technological solutions of other people,
- we would like to attain a freedom to apply new technologies and solutions which did not exist at the time of the creation of the core elements of previous systems.

Computing in general, and specifically high-performance computing, requires fast computers. Users may try to use personal computers, but their machines may turn out to be insufficient for computationally-demanding algorithms. Supplying a fast computer for each person in the academic or corporate world is associated with increased costs. Personal computers are probably not in use all the time, so using the idle cycles of these machines on-demand to speed up computations might be considered. However, a better idea may be to use computing centers which provide much-needed resources. Auto-scaling in such a system is extremely important. When there are no resources for executing jobs, they are enqueued to wait for idle machines. The processing of each job takes some time; thus, when the user submits jobs too frequently, the queue may grow too big and jobs may not be executed within a satisfactory time frame. This means more processors are needed. On the other hand, there may be no jobs to execute during certain periods of time. This means that machines will be idle, thus producing undesirable costs. These problems can be solved by using clouds to perform BoT computations. The system under discussion is aimed at providing the running machines when needed and withdrawing them when idle. Another aspect of the research is to provide a flexible support for storing the input data meant for processing as well as the output results of the processing, and to tune the data storage when a necessity emerges.

4. Concept

One of our main goals is to create a system capable of processing huge quantities of data coming in from a network of sensors by the Internet. In this case, the system will collect information from sensors, create necessary resources, perform computations, and provide results for consumers. Another possible use is a scenario where tasks to be executed are added by the user. In this case, the user needs to upload their programs and input data, and then add tasks to the system queue. The system will automatically find resources capable of executing the given tasks, dispatch them to the appropriate machines, and place the results into data storage. We assume that many jobs will be similar. For example, data coming from different sensors will be processed in the same way, but with different inputs. The user also might use their programs multiple times with different inputs, or perhaps share their programs with some community (for example, team members). To enable this, we assume the use of repositories which store user programs (for example, written in Python or Java) and input data.

Our system's objectives are as follows:

- Processing of huge quantities of data:
  - Tasks can be run periodically (useful in processing the data incoming from a network of sensors).
  - Tasks can be run on demand (useful in executing user programs and processing the data uploaded by users).

- Resource usage monitoring, optimization, and auto scaling:
  - Awaiting tasks are dispatched to awaiting machines.
  - If the system is able to process the awaiting tasks with fewer machines than are currently turned on, the idle machines are turned off.
  - If the system queue grows too long, new machines are provided.
- Scalability.
- Easiness-to-install on common machines /-adapt to the existing architecture.
- Use of virtual machines:
  - Enable the executing of various user programs.
  - Enable easy system upgrade.
- Ability to implement and run different Load Balancing algorithms:
  - This allows us to measure the efficiency of different Load Balancing algorithms.
- Providing repositories for user programs, the input and output data.
- Web interface.
- Support for easy-to-create system backup.
- Providing authorization.

To meet the above requirements based on our research, we decided to exploit the master-slave paradigm in cloud resources with agent-controlled distributed computing nodes and distributed-data storage combined with a centralized database (to store the system's entities).

5. Description of the system

Below, we describe our system, predict and analyze some potential issues, provide solutions, and highlight the design decisions we made. In Section 5.1, we concentrate on the system's design, which is followed by an example scenario. While Section 5.3 is focused on the system's architecture, Section 5.4 presents the use of an agent-based approach to the functionality of the system under discussion. Section 5.5 is aimed at data-related issues of the system.

5.1. Design

In our effort, we exploit solutions known from previous research while enhancing them with our own approach related to new, promising trends. In comparison with solutions known since the 1990's, our system features new user-oriented cloud-bound facilities like turning on/off VMs, adding new machines, storing user programs, input and output data in repositories, convenient data processing from the perspective of the user, and data sources (e.g., sensors).

Data processing requires a lot of resources, but the intensity of the related jobs is assumed to be non-constant – for example, sensors may send data once per day,

after the collection of information is completed. This implies that the system will need to enqueue and process a lot of jobs in one period of time and will be idle in another. As a solution, we decided to create a component called Load Balancer, which will use AWS SDK for Java [2] to dynamically create and terminate virtual machines and use them to execute bag-of-tasks. By using virtual machines, the system is able to execute almost every algorithm (program) implemented in a number of popular languages (provided it is possible to be executed on a given machine). Our Load Balancer creates VM instances whenever necessary and terminates them when they are idle. Load Balancer can create a large number of VMs – however, it is not a good idea to connect all VMs directly to Load Balancer and measure their loads, since thousands of VM instances may be processing jobs and many sensors may be sending huge amounts of data.

To solve the issue of potentially too many connections to Load Balancer, we use a hierarchical structure consisting of masters and slaves, with one distinct root unit. Load Balancer creates machines of two types: Commander and Worker. Commander is an agent playing the role of Master w.r.t. Workers. It measures the load of each Worker (which is a Commander's slave) and dispatches jobs between them. Load Balancer is a root unit which is superior to Commanders and distributes jobs among them. Single Load Balancer controls many Commanders (within reason) while each Commander controls many Workers. The main advantage of such a structure is the reduction of connections to one machine and, thus, better system scalability.

User programs, jobs inputs, and results need to be stored. It is inappropriate to store every file on the same machine due to a huge amount of data being transferred between system parts. Such storage should be distributed, and the client or user should not have to know which machine will be the best for storing their files. We solve this problem by introducing File Storage Units (FS Units) and File Storage Service (FS Service). The principal responsibility of FS Units is to store files, while FS Service measures their load, and on this ground provides information on which File Storage Unit is stores each file.

To provide consistency between the system's modules, we decided to exploit a database-like approach. In this database we store the most important information about jobs and user programs. This approach provides two main advantages: 1) transactions help keep the system synchronized, and 2) in case of failures, databases keep the system's snapshot (making it easy to create a backup). There is also one disadvantage: this part is centralized – however, we made some improvements to reduce the number of connections, queries, and the size of stored data.

The system is meant to provide a user-friendly interface. Humans can communicate with the system via a Front-end accessible through an Internet browser. Machines can also use it (for example using the `htmlunit` library for Java), but it is easy to implement an appropriate proxy for them to access the system's functionality.

5.2. Example scenario of job's execution

1. The user first needs to create a package with their program using a tar.gz archive containing a bash script named run.sh. In this example scenario, the user's program will be a two-dimensional interpolation written in Python. The algorithm will read a path to a file that contains two-dimensional interval boundaries from standard input and produce a function interpolation to be displayed.
2. The next step is to add the user's program to the system via Front End. The first part is to add the user program entity to the system, and program files will be uploaded in thereafter. In this step, the user needs to provide the program's name and some additional information (for example, if the program is private or available to the public [can be accessed by other users]).
3. Once the program entity is created, the user need to upload a previously-created package containing the program (see point 1). The system will automatically redirect the user to the proper File Storage Unit.
4. Once the package has been uploaded, the user's program is ready for use. There is an ability to create a job for this program via Front End. The user provides a job name and chooses a user program for which the job will execute.
5. After having created the new job, the user uploads the job's package (package containing the job's input data). In this scenario, we need to provide a file named config.ini. In this file, we store boundaries for the two-dimensional interpolation interval. This is only an example the user will choose a suitable input data format.

The content of file used in example: config.ini
[Dimensions]
xleft = 0
xright = 4
yleft = 0
yright = 4

6. The job is now ready to be processed, but it might not be executed immediately. Load Balancer's thread first calculates the load and provides the necessary resources. Creating new VM instances may take some time.
7. The job is dispatched between Commanders; i.e., it will be assigned to a Commander.
8. Commander delegates the job to one of its Workers.
9. A request for the user's program and job files is enqueued in Worker's download queue.
10. Worker processes its download queue and downloads the requested files.
11. Once all the necessary files are downloaded, Worker enqueues the execution of this job.
12. Worker executes the job.

13. After a successful execution, Worker packs the results into a folder as a tar.gz archive and uploads it to File Storage Unit.
14. Results are available for downloading via Front End.

5.3. Architecture

The system that implements the above concept is comprised of modules that handle the incoming tasks, data, and interactions with the user (please see Fig. 1):

- Load Balancer.
- Commander.
- Worker.
- Database.
- File Storage Unit.
- File Storage Service.
- Front-end.

As mentioned above, the idea is to create an experimental hybrid architecture which combines the advantages of the centralized and distributed approaches. The modules involved in the centralized part of the system are Load Balancer, Database, File Storage Service, and Front-end. The modules which belong to the distributed part are Commander, Worker, and File Storage Unit. Both approaches feature some advantages. The distributed approach offers high scalability, huge computing power, module responsibility reduction, faster data transfer (with a large number of connections), and improved fault-tolerance. The centralized approach is easy to synchronize, as it features lower latency, easier error detection and system-backups creation, and is easier to secure (e.g., centralized authorization).

To balance dependencies between the system units, avoid communication flooding, increase scalability, and balance the module load, we decided to create a hierarchical structure consisting of masters and slaves: Commanders are supervisors of Workers and Load Balancer is a supervisor of Commanders. A similar approach was adopted in File Storage: File Storage Service is a supervisor of File Storage Units. Each supervisor monitors the load of its slaves. With this information, it can determine the most appropriate machine for a given job and, as a result, balance the load of the assigned slaves. For the distributed modules responsible for executing jobs, we decided to use cloud resources and apply an agent-based approach. Thus, Commanders and Workers are agents running on virtual machines inside the cloud (e.g., EC2). They are created by Load Balancer when demand for computing power grows and terminated when demand drops. The other system parts are static (they are online all the time) and are functioning outside the cloud (running on "standard" servers).

5.4. Use of agents

In order to provide a mechanism for allowing instances to interact, the system is organized as a set of agents which are assigned specific roles in the system [17]. This

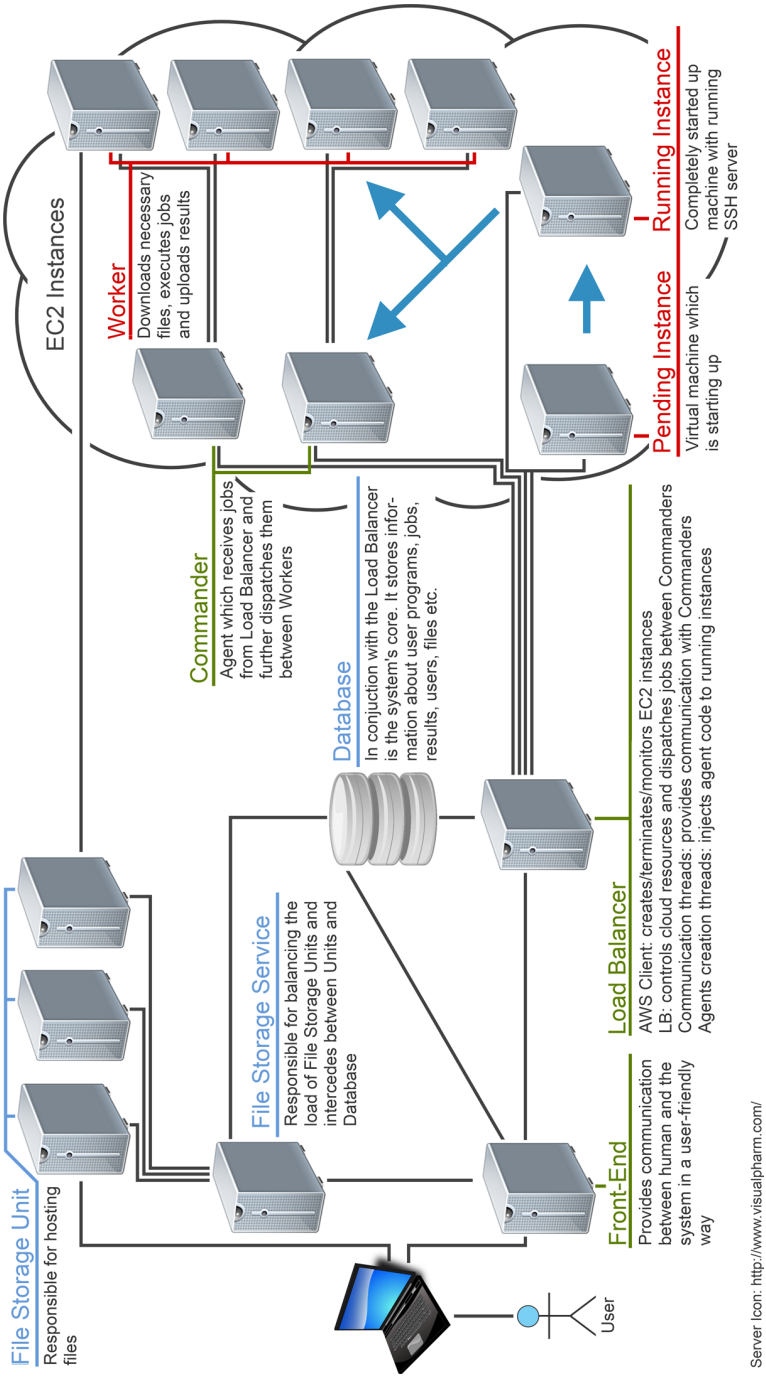

Figure 1. Architecture of agent-based scalable system of executing bags-of-tasks.

helps better achieve the system's goals. The system exploits a number of mechanisms to solve the issues of interaction, negotiation, cooperation, and control of a large number of machines running multiple processes. While all of the modules in the system can be considered agents, reducing the notion of agents to two types of program units may help simplify matters. These types are as follows:

From the Load Balancer node, jobs are dispatched between agents. As an agent, we consider a virtual machine (called "instance" in EC2) which runs one of the following programs: one that *executes a job*, or one that *dispatches jobs*. In our system, we distinguish between two kinds of agents: Workers and Commanders. Workers are designed as nodes which are the final consumers of jobs and the producers of job results. Workers also report their CPU load to Commander and Load-Balancer. Commander acts like a kind of supervisor over Workers: they dispatch jobs between them and dispatch their messages to Load-Balancer. Each Worker must be connected to Commander, and Commander must have a Load-Balancer node connected to it. Load Balancer is continuously collecting data and processing information about jobs, the number of agents, and their respective loads. Whenever necessary, it creates new agents or terminates idle ones. More details on the issues of scaling can be found in Section 6.

Workers and Commanders usually run on machines provided by a cloud (in the current implementation, we use Eucalyptus [6, 7], which is a private cloud-computing platform that implements the Amazon specification for EC2, S3, EBS, and IAM). Whenever cloud services are unavailable, or when it is necessary to have continuously-running machines (for example, when jobs need to be executed immediately, without waiting for a virtual machine to initialize), it is possible to use statically-dedicated agents; these agents are physical machines outside of cloud services. In order to use them, the administrator needs to provide Static-Commander's IP addresses to Load Balancer, and Static-Workers should be connected to a running Commander instance or Static-Commander. Load Balancer uses statically-dedicated agents in the same way as cloud-service agents, but it cannot terminate them when they are idle.

The life cycle of each EC2 instance is continuously monitored. The most important states of the instance are: pending, running, and shutting-down. Load Balancer controls the status of each instance to check whether the agents are working properly. It is especially helpful for the administrator when the system is using a considerable number of cloud resources and/or the resources are not physically accessible. In our system, the agents store local data only until they are terminated. When a virtual machine starts again, all of the previous data which was saved on the instance is lost. To store files, we created our own storage facility (called File Storage), introduced in Section 5.1.

5.5. Data Storage

In our system, we distinguish between two types of data. The first type is related to the objects which we use for implementation purposes. Some of them (entities) need to

be persisted: information on users, their programs (owner, access rights, URL, etc.), and jobs. This kind of information does not require a lot of space, but it is frequently used by the system components and needs to be consistent. We use a database to store such data. Due to the use of transactions, the information is consistent. Moreover, it is easy to create a system backup to increase fault-tolerance. Along with all these advantages, there is one problem – this part of system is centralized. We overcome this problem by reducing the number of connections and queries, so the common server should be able to provide a proper responsiveness level. The second type of information is related to files such as user programs' files (sources, binaries), input data, and results. All of these files cannot be stored together on a single machine (as explained in Subsection 5.1). At the beginning of executing a job, each Worker needs to download relevant files and to upload results once the execution is finished. To make it fast, a high-speed connection is required. Another constraint is disk operation speed. For these reasons, it is better to use multiple servers to store data. We implement it by providing an own File Storage system consisting of File Storage Units and File Storage Service, introduced in Section 5.1. Implementation details of File Storage modules can be found in Section 7.

6. Scaling, load balancing, and performance issues

In order to keep up with the changing amount of data and the number of jobs, we need to provide a scalable environment for agents. The first problem is balancing the number of running machines. As mentioned above, there can be a large number of jobs to execute at one period of time, whereas there may be nothing to do in another period. This means that keeping all the agents continuously running is useless in some situations. This aspect is tackled by Load Balancer, which can create or terminate VM instances (which are running agents). The number of instances to be created or terminated depends on the number of jobs that may be running on Workers in parallel, the maximum number of Workers assigned to one Commander (these variables are provided by Load-Balancer), and the number of jobs enqueued.

Our approach to the issue of load balancing follows a hierarchical structure. To avoid a flood of communication flood, we decided that only Commanders are allowed to communicate directly with Load Balancer, in order to lower the number of connections and requests which may generate high load. The Load Balancer node in its job dispatching phase considers only Commanders as the components where the jobs can be delegated. Such an operation is not trivial since it is difficult to foresee how load will change over time. Let us consider an example: there are three Workers in the system. Two of them are executing jobs (one job per each Worker), and their CPU usage is above 90%, while one Worker is idle. There are ten jobs to dispatch. The Worker, which executes a job, first needs to download the required files, such as a user program and input data. If the load balancing algorithm considers the current load only, all of them may be dispatched to one (idle) Worker, because downloading the necessary files does not generate a big load (CPU), but may take some time. As

a result, two Workers will finish processing the jobs assigned to them and become idle, while the third Worker will handle ten jobs. The main objective of our algorithm is to balance the load proportionally for each Commander, e.g., to match the average load. In our calculations, we take into account job queue length, download queue length, memory load, and CPU usage.

Below, we present the first version of the algorithm we use to dispatch jobs to Commanders. This is only an example, since our research was not primarily focused on finding the best load-balancing algorithm (at the moment, our concern is related to the architectural and scalability issues and making the whole system work reliably). In the future, we are aiming to implement other algorithms and provide a mechanism which will allow us to configure (choose) an algorithm which will be used in the system to provide the best attainable efficiency for any given conditions.

Detailed job dispatching algorithm:

1. For each $\mathbf{Commander}_i$ calculate its $\mathbf{load}_i$, taking into account the download queue, job queue, processor, and memory load – they have assigned weights and are summed to form an integer value. It is similar to the weighted mean, but without dividing by the sum of weights (not necessary in our case). For example, each job in the queue has weight 2, and 100% of CPU and memory usage has weight 5. If Commander uses 40% of its CPU, 60% of memory, and has 3 jobs in queue, then its load is $0.4 \times 5 + 0.6 \times 5 + 3 \times 2 = 2 + 3 + 6 = 11$.
2. For each $\mathbf{Commander}_i$ set $\mathbf{newJobsAmount}_i$ to 0 ($\mathbf{newJobsAmount}_i$ is the number of the new jobs which will be dispatched to $\mathbf{Commander}_i$ in the next steps of this algorithm).
3. For each job to dispatch find $\mathbf{Commander}_i$ with the lowest $\mathbf{load}_i$, increase its $\mathbf{newJobsAmount}_i$ by 1 and increase its $\mathbf{load}_i$ considering just the added job as a job in the queue. Store these values in a map named **commandersNewJobs** defined as Map<Commander, Integer>, where $\mathbf{Commander}_i$ is associated with its number of new jobs.
4. Sort the map in descending order by values.
5. For each $\mathbf{pair} = (\mathbf{commander}, \mathbf{newJobsAmount})$ in **commandersNewJobs**:
    1. Check if there are any jobs to pick.
    2. For $\mathbf{k} = 1$ to **pair.newJobsAmount**:
        1. If (**awaitingJobs.size** > 0):
            1. **Job j** = **awaitingJobs.pop**();
            2. **pair.commander.dispatch**(j);
6. If there are still awaiting jobs go back to 1.

To improve the system's performance, we implemented a multi-thread mechanism which allows Worker to download the required files (for example, those needed to execute a subsequent job) while it executes other jobs.

For balancing the load of data storage machines, we use File Storage Service – this controls each File Storage Unit. Each Unit monitors its files – when some change

occurs, it informs the Service about it. This information is used by the service to choose a machine which will store a new file. It is not trivial to choose the most relevant machine; for example, if data size is the only consideration, a file of 1 GB size would approximately equal to ten 100 MB files. If there are 2 File Storage Units in the system, a 1 GB file would be stored on the first Unit with ten 100 MB files stored on the second. There can be a situation where these 10 files are likely to be downloaded, while 1 GB file is downloaded only once a year. This means that the first machine would be idle most of the time. To improve matters, the service considers the number of files stored on a machine as well. To calculate the load in the first system's version, we use the weighted mean. In the future, we aim to add a further parameter – the popularity of machine (or that of each file), which will take into account the number of download requests for each file. When the data transfer speed is slow and the number of uploads and downloads is high, this implies a necessity to provide a further server. Scaling the Storage Service is quite easy – the Administrator just needs to register a new Unit in the Service. It can be done at any time, and does not require a system restart. In case of a large number of users, it is possible to run multiple Front Ends. With some implementation changes, it is possible to run multiple instances of Load Balancer, with the requirement that the agents assigned to one Load Balancer will be hidden from another one. This is possible, as these Load Balancers would use the same Database which provides synchronization, e.g. due to its possibly transactional nature.

7. Implementation details

Below, we provide some details on the implementation of the system's modules.

Cloud resources: Eucalyptus [6, 7], which we use in our implementation, is a private cloud-computing platform that implements the Amazon specification for EC2, S3, EBS, and IAM. It provides an Amazon Web Services (AWS)- compliant EC2 based web service interface for interacting with the Cloud service. To control cloud resources from within the application, we use AWS SDK for Java [2], which provides Java APIs for many AWS services. Most important to us was the ability to control EC2, which can be achieved by using AmazonEC2Client included in this SDK. We also use Euca2ools [5], which are command line tools for interacting with Amazon Web Services (AWS) and other AWS-compatible web services such as Eucalyptus and OpenStack.

Load Balancer: We implement Load Balancer using Java. Communication with Front-end is provided using a web service written with Apache CXF [4]. Dependency injection is provided with Spring-framework. For executing periodic procedures, we exploit the Quartz library [10]. Currently, we are scheduling two periodic procedures – `refreshAgents` and `refreshJobs`. The `refreshAgents` procedure is responsible for scaling the cloud environment, whereas the `refreshJobs` procedure is used to pick recently-submitted jobs from the database and dispatch them between commander nodes. We use AWS SDK for Java [2] to communicate with cloud services (creat-

ing, terminating, and monitoring VM instances). The total number of jobs, and the number of agents that are currently running on instances, are used for calculating the amount of cloud resources to create or terminate. Upon connection to an agent instance, Load Balancer creates an agent object for it and stores this object in the map. For each Commander, Load Balancer runs two threads – one for reading messages and one for sending. Communication with Commanders is based on java.net TCP sockets, yet messages are JSON encoded [9]. In order to inject code for the agents, we needed to use SSH/SCP/SFTP API – for supporting this functionality, we utilize the `sshtools` library.

Messages: In order to distinguish between types of operations, we implemented our own message protocol. Each message is encoded: [4 bytes – size of message][1 byte – operation code][(Size – 1) bytes – operation data encoded in JSON]. The first 4 bytes are message bytes with little-endian encoding. The next byte is an operation code. To execute operations, we use the strategy pattern [19]. We created a strategy interface which delivers the `execute(String operationData)` method. Whenever the node receives a message, it uses the strategy assigned to an operation-code and calls the `execute()` method with a given operation's data.

Use of the Strategy pattern:

For example a **message** with prefix "L" is received:

1. If **(this.strategies.containsKey["L"])**:
    1. **Strategy s = this.strategies.get("L")**;
    2. **s.execute(message)**;

A strategy assigned to "L" is being executed. It performs a load update for Commander's Workers:

1. Map string **message** to **LoadReport** class:
   r = objectMapper.readValue(message.substring(1),LoadReport.class);
2. Fetch from **LoadReport** list of **Workers Loads**:
   workerLoads = r.getWorkerLoads();
3. Get all **Workers** assigned to **Commander**:
   workers = executionThread.getCommander().getLoadBalancer() .getWorkers();
4. For each w in **workerLoad**:
   workers.get(w.getWorkerId()).updateLoad(w);

Commander and Worker: In case of cloud nodes, we use the Python language (which delivers an elastic and light API). When virtual machines do not contain a pre-installed agent code (Command or Worker), we need to send the files during the agent code injection phase; therefore, the code package should be as light as possible. Another advantage is that the code does not need to be compiled; since most Linux distributions have Python installed, our agents should work on nearly every machine under Linux. We also had to face constraints in the predefined virtual machine images, such as Java Virtual Machines not being installed. Due to these reasons, Commander and Worker code are written in Python. In order to execute

a job, Worker needs to download its files and, upon execution, upload the job's results. Each request to File Storage Unit is AES encoded; therefore, Worker additionally uses the PyCrypto module. Worker runs threads for downloading, reading, and relaying messages from/to Commander.

Data Storage: Data Storage involves two module types: File Storage Unit and File Storage Service. File Storage Service is implemented as a web service which measures File Storage Units loads and executes their queries to Database. For example, when a job is finished and the results are uploaded, Unit sends this information to Service and, later, it changes the job's status in Database. Service does not execute queries immediately – it collects requests and cyclically executes all enqueued queries. This approach allows us to reduce the number of connections to Database. File Storage Service is implemented in Java using the Spring-framework; CXF is used to provide the web service, Hibernate – to provide database operations, and Quartz – to provide job scheduling. File Storage Unit, which stores files, provides a user-friendly GUI accessible by any browser. It uses the Spring-MVC framework with models and controllers implemented in Java and views written in Freemarker. Additionally, it uses javax.crypto for processing AES-encoded requests. It also reports its load to File Storage Service with SOAP, using a CXF client.

Front-end: Front-end provides a user-friendly GUI accessible by any browser; therefore, it is implemented in Java using Spring-MVC with views written in Freemarker. With Front End, the user can manage user programs and jobs. These are stored in the database, and operations are developed with Hibernate. The administrator can manage instances, e.g., monitor or kill them and steer the system's load. Communication between Front End and web services (provided by Load Balancer and File Storage Service) is implemented using CXF (both clients on and endpoints).

8. Evaluation

In our tests, we are using the Eucalyptus Community Cloud [6] and cloud resources provided by FutureGrid [8] running Eucalyptus (version 3). We were provided with Cloud-Based Support for the Distributed Multiscale Applications project [3] with access to two clusters:

1. india (IBM iDataPlex at IU),
2. sierra (IBM iDataPlex at SDSC).

The following virtual machine types were provided: *small* – 1 CPU, 512 MB of RAM, 5 GB of storage space, *medium* – 1 CPU, 1024 MB of RAM, 5 GB of storage space, *large* – 2 CPUs, 6000 MB of RAM, 10 GB of storage space, x-large – 2 CPUs, 12,000 MB of RAM, 10 GB of storage space, x-large – 8 CPUs, 20,000 MB of RAM, 10 GB of storage space. Yet, we did not use the most powerful virtual machines due to the undemanding computations. Most of the time, we used the medium type of virtual machine.

During development and tests, we used the medium type of virtual machines which proved sufficient for our computations. In practical work, the number of re-

sources was limited – it was possible to create only up to X instances, where X is constrained by cloud administration. We were also limited by the network communication bandwidth, which may affect job execution. Finally, the system has been tested on a configuration where all non-cloud modules were running on a single shared Virtual Machine, and cloud resources were delivered by the FutureGrid project. This implies that our test environment was comprised of one virtual machine (which was running Load Balancer, Front-end, Database, File Storage Units, and File Storage Service) and a few machines from cloud resources provided by the FutureGrid infrastructure (which were running Commanders and Workers).

To show real-time usage of the system, we provided three examples: Two-dimensional function interpolation, Noise Filtering, and a sample benchmark.

8.1. Two-dimensional function interpolation

A job implemented in Python (with the use of *sci-py*, *num-py*, and *matplotlib*) was supposed to interpolate a function using the RBF algorithm. In order to perform this interpolation in parallel, we divided a 2-D area into four separate sub-areas and dispatched, as jobs, each worker executing a different sub-area. The left display (please see Fig. 2) depicts a result of executing this algorithm on the mentioned four two-dimensional intervals (sub-areas).

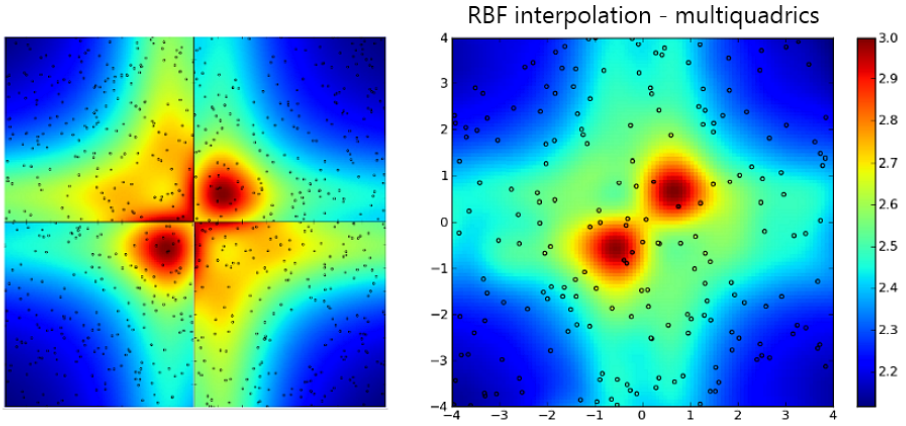

Figure 2. Result of example jobs: distributed two-dimensional RBF interpolation in four separate areas (on the left) and non-distributed two-dimensional RBF interpolation in one cohesive area (on the right).

On the right display, we can see the result of interpolation of the same function, but evaluated in only one area. The most significant differences can be observed near point (0,0) – a reason for that is a greater density of interpolation points in this area when executed in parallel than when executed in the one cohesive area. Another factor that disturbs the computation result (in the case of parallel computations)

is area decomposition – the function did not have a neighbor point from which to interpolate, so the computation was affected. This stems from the embarrassingly-parallel nature of the bag-of-tasks paradigm. Nevertheless, this particular example demonstrates that not every kind of computation can be reasonably parallelized in the BoT paradigm.

8.2. Noise filtering

Image processing appears to be a more-suitable candidate for the BoT paradigm. Each instance of this problem does not require additional information from other instances for processing. A job implemented in Python was supposed to denoise the provided image using the TV Denoise algorithm. For a parallelized task, the image has been broken down into pieces, with each worker processing a different piece. In this particular example, we used a moon-landing photograph as an original image and added some noise to this image with Gimp 2.0 (which resulted in a noisy image). Figure 3 shows the compiled results of noise filtering. We zoomed in and sharpened the highlighted area a bit in order to compare the unfiltered image with the filtered one.

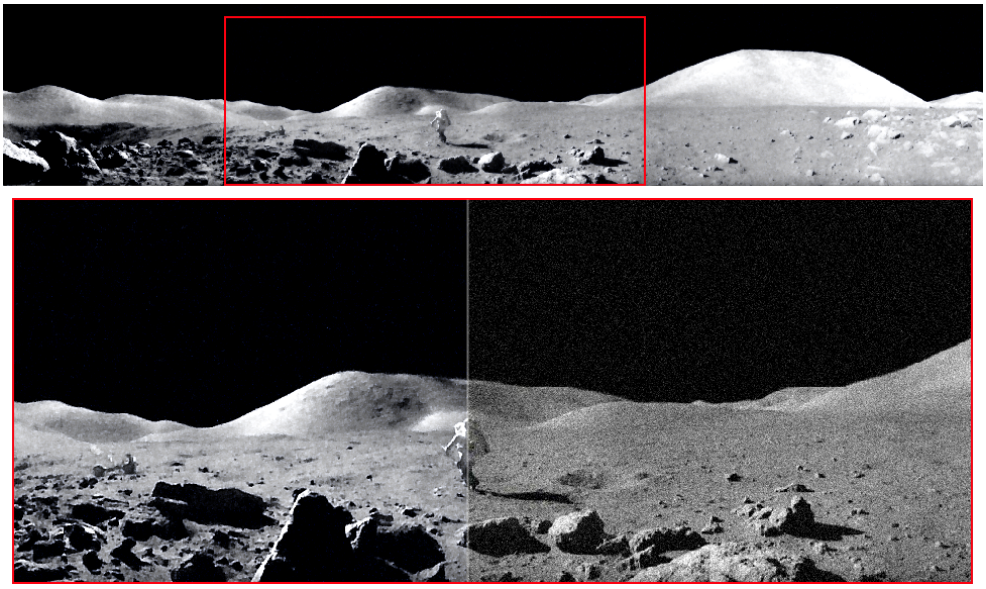

Figure 3. Compiled results of parallel noise filtering. Highlighted area shows the difference before noise filtering (right side) and after noise filtering (left side). Credit: Apollo 17 Crew, NASA; Mosaic Assembled & Copyright: M. Constantine (moonpans.com).

We evaluated the performance of the application on a set of different-sized images. Each image from this set was processed by $N = \{1, 2, 3, 4, 6, 8, 12, 16\}$ nodes. One can readily see that the execution makespan drops as the number of nodes grows. In

some cases, it was impossible to load the image into a Worker's memory due to the large raw-data (decompressed) size. Execution times are presented in Figure 4.

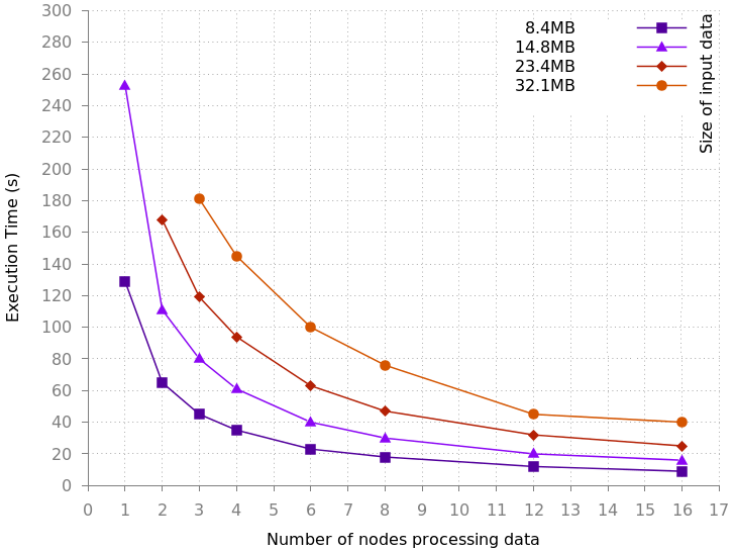

Figure 4. Parallel noise – filtering execution times for various sizes of data sets.

8.3. A sample benchmark

In order to test the efficiency of virtual machines, scaling algorithms, and load balancing algorithms, we implemented a benchmark job. The idea was to use as many virtual machine resources as possible. This job creates N processes, and in each, it multiplies two 2500×2500 arrays. First, we created one job with 6 processes, then two jobs with 3 processes, the next three jobs with 2 processes, and in the end, six jobs with 1 process. It should be noted that a job is delegated to a Worker; so, if processes are encapsulated in a single job, they will be executed on a single Worker. To maximize the efficiency, independent processes should be run in separate jobs. Execution times are presented in Figure 5.

We can observe a significant difference between benchmark execution time with only one node and two nodes; this is due to the fact that the benchmark job requires too much memory to allocate on a single machine at once. This causes delays between executing the multiplication operations in each of 6 processes, because each process has to wait for memory to allocate. With three processes in a job, sufficient memory size allows for faster execution, since less machine time is spent on allocating and waiting for free memory in which to allocate the array. Differences in execution time are not so pronounced when the number of agents ranges from 3 to 6. The computation time lessens, but it is still affected by the time of job propagation between

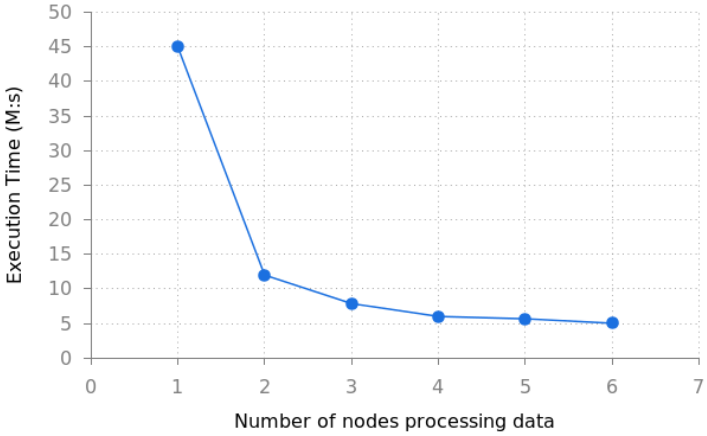

Figure 5. Benchmark-execution time in function of the number of agent instances used in our system.

nodes as well as the time spent on scaling cloud resources (Load Balancer cyclically creates new machines, and the pending of each instance also consumes some time).

9. Conclusions and future work

In the scope of building a system for executing bag-of-tasks applications, we focused mainly on scalability issues, exploring the problem of executing bag-of-tasks in clouds, studying the suitability of existing technologies in our solution, designing the architecture, and creating the system's prototype. As a result, we presented a prototype of an agent-based, hierarchical system for executing bag-of-tasks in clouds. The system offers a dynamic allocation of resources, and seeks to find an optimal resource allocation based on the load-balancing algorithm implemented. Moreover, the system can execute any computation. provided it is possible to run its code in terms of the installed software on given machines (workers).

The system provides the following features:

- Execution of bag-of-tasks applications.
- Data storage (entities, binary files, sources, input data, and results).
- VM-instances monitoring.
- Load balancing for Commanders, Workers, and File Storage Units.
- Creation of VMs when they are necessary and terminates them when idle.
- Dynamic scaling.
- User authorization and basic security features.
- Easy-to-create backup.
- Support for user programs written in various languages.

- Easy upgrading of VM instances.
- User-friendly web interface.
- Ability of working with data coming from machines or other data sources, e.g., sensors.

The system has been tested in a configuration where all non-cloud modules were running on single shared Virtual Machine, and cloud resources were delivered by the FutureGrid project. In this environment, we have conducted integration tests. The most important was to test the job's execution scenario described in Section 5.2 of this paper. Other tests were aimed at checking the correctness of executing the multiple jobs, file transfer, management of cloud resources, and system administration. All the tests have been successfully accomplished. We have also tested three examples of bag-of-tasks more thoroughly, described in Section 8: 2-D function interpolation, Noise Filtering, and a sample benchmark.

As expected, the two-dimensional function interpolation example showed that not every problem can be easily parallelized. In this case, the result of the parallelized computations performed according to the bag-of-tasks paradigm differs from the result of computations performed without parallelization. This is mainly due to the lack of information sharing between tasks working on the same problem while analyzing another set of data. Noise-Filtering parallelization provided much better results because, in this case, the mentioned lack of information sharing was not critical. In this case, the execution makespan drops as the number of nodes grows (Fig. 4). A similarly-decreasing execution time related to the increasing number of nodes was observed in the case of the benchmark (Fig. 5).

Based on our experience with the system on a small scale, we will gain a better understanding of how to cope with its adaptation to a large scale. Surely, the most important work to be done is to test the system in an environment similar to an operational one. First, we plan to use some kind of a cloud simulator, and then test our architecture in an environment in which each module will be running on a dedicated machine (and involve a larger set of cloud resources). An ideal solution would be to create our own cloud and adapt it to system requirements. The cloud resources in which we had access to were not stable – cloud services' unavailability and malfunctions frequently occurred. We often encountered the "infinity pending" instance problem, which means that an instance is in a consistent "pending" state – a solution to this problem was to reboot or terminate such a virtual machine after some time had elapsed, and (if it was finally terminated) to create a new one. What is worse, the cloud configuration often changed during the development phase, which implied a need to implement some new solutions. For example, sometimes a connection was rejected – it was dependent on cloud security settings and the agent-instance launch time.

However, small-scale tests have helped us to find some potential improvements. To eliminate the complicated source injection into instances, we are planning to create our own virtual machine image which could be optimized with respect to the execu-

tion of jobs. Such images should have pre-installed Worker's code, Commander's code, and the most-common libraries (which now need to be downloaded to make the execution of jobs possible). We also aim to install Java on such machines, implement a Java version of agents, and replace message passing by Java Message Service (JMS). Currently, the system exploits only one load-balancing algorithm (as described in Section 6). In the future, we plan to make it possible to choose the algorithm which will be exploited by Load Balancer for particular load situations and test them with an emphasis on performance. To increase scalability and the ability to handle more requests, we intend to consider the use of clustering and sharding: JBoss Clustering for webapps and an NoSQL database (e.g. MongoDB). Also, improvements in the File Storage system's part are envisioned. We are currently working on a component whose functionality is aimed at storing files under heavy loads.

Acknowledgements

The authors are grateful to prof. K. Cetnarowicz for comments on agent-based research. Our thanks go to Mr Tomasz Bartynski for fruitful cooperation. We also want to appreciate the delivering of cloud resources by FutureGrid `https://portal.futuregrid.org/`.

This research is partly supported by the EU ICT-FP7-269978 Project VPH-Share and the European Union within the European Regional Development Fund program as part of the PL-Grid PLUS Project (`http://plgrid.pl/plus`).

References

[1] *Amazon EC2 website*. `http://aws.amazon.com/ec2`. Access: 12.09.2013.

[2] *AWS SDK for Java*. `http://aws.amazon.com/sdkforjava`. Access: 12.09.2013.

[3] *Cloud-Based Support for Distributed Multiscale Applications*. `https://portal.futuregrid.org/node/937/project-details`. Access: 17.09.2013.

[4] *CXF website*. `http://cxf.apache.org`. Access: 23.08.2013.

[5] *Euca2ools*. `http://www.eucalyptus.com/docs#euca2ools`. Access: 17.09.2013.

[6] *Eucalyptus*. `http://www.eucalyptus.com`. Access: 17.09.2013.

[7] *Eucalyptus and AWS*. `http://www.eucalyptus.com/aws-compatibility`. Access: 17.09.2013.

[8] *FutureGrid website*. `https://portal.futuregrid.org`. Access: 17.09.2013.

[9] *JSON website*. `http://www.json.org`. Access: 23.08.2013.

[10] *Quartz Scheduler website*. `http://quartz-scheduler.org`. Access: 23.08.2013.

[11] Ambroszkiewicz S., Cetnarowicz K., Kozlak J., Nowak T., Penczek W.: *Modeling Agent Organizations*. In: Intelligent Information Systems, *Advances in Soft Computing*, vol. 4, pp. 135–144. Physica-Verlag HD, 2000. ISBN 978-3-7908-1309-8. `http://dx.doi.org/10.1007/978-3-7908-1846-8_13`.

[12] Anglano C., Canonico M.: *Scheduling algorithms for multiple Bag-of-Task applications on Desktop Grids: A knowledge-free approach.* In: *IPDPS*, pp. 1–8. IEEE, 2008.

[13] Antonis K., Garofalakis J., Mourtos I., Spirakis P.: A hierarchical adaptive distributed algorithm for load balancing. *J. Parallel Distrib. Comput.*, 64(1): 151–162, 2004. ISSN 0743-7315.
http://dx.doi.org/10.1016/j.jpdc.2003.07.002.

[14] Balis B., Kasztelnik M., Bubak M., Bartynski T., Gubaa T., Nowakowski P., Broekhuijsen J.: The UrbanFlood Common Information Space for Early Warning Systems. *Procedia Computer Science*, vol. 4, pp. 96–105, 2011. ISSN 1877-0509. http://dx.doi.org/http://dx.doi.org/10.1016/j.procs.2011.04.011. In: *Proceedings of the International Conference on Computational Science*, ICCS 2011.

[15] Beaumont O., Carter L., Ferrante J., Legrand A., Marchal L., Robert Y.: Centralized versus distributed schedulers for multiple bag-of-task applications. In: *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pp. 24–24. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 1-4244-0054-6.

[16] Bruno J., Coffman Jr. E.G., Sethi R.: Scheduling independent tasks to reduce mean finishing time. *Commun. ACM*, 17(7): 382–387, 1974. ISSN 0001-0782. http://dx.doi.org/10.1145/361011.361064.

[17] Cetnarowicz K.: From Algorithm to Agent. In: *Proceedings of the 9th International Conference on Computational Science*, ICCS 2009, pp. 825–834. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-01972-2.
http://dx.doi.org/10.1007/978-3-642-01973-9_92.

[18] da Silva F. A. B., Senger H.: Scalability limits of Bag-of-Tasks applications running on hierarchical platforms. *J. Parallel Distrib. Comput.*, 71(6): 788–801, 2011. ISSN 0743-7315. http://dx.doi.org/10.1016/j.jpdc.2011.01.002.

[19] Gamma E., Helm R., Johnson R., Vlissides J. M.: *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, Reading, Mass. [u.a.], 1994. ISBN 0-201-63361-2.

[20] Geelan J.: Twenty one experts define cloud computing. *Cloud Computing Journal*, http://virtualization.sys-con.com/node/612375, 2009.

[21] Gutierrez-Garcia J., Sim K. M.: Agent-Based Service Composition in Cloud Computing. In: *Grid and Distributed Computing, Control and Automation*, T. H. Kim, S. Yau, O. Gervasi, B. H. Kang, A. Stoica, D. lzak (Eds.), *Communications in Computer and Information Science*, vol. 121, pp. 1–10. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-17624-1.
http://dx.doi.org/10.1007/978-3-642-17625-8_1.

[22] Gutierrez-Garcia J. O., Sim K. M.: A Family of Heuristics for Agent-Based Cloud Bag-of-Tasks Scheduling. In: *Proceedings of the 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, CYBERC'11,

pp. 416–423. IEEE Computer Society, Washington, DC, USA, 2011. ISBN 978-0-7695-4557-8. http://dx.doi.org/10.1109/CyberC.2011.74.

[23] Ibarra O. H., Kim C. E.: Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *J. ACM*, 24(2): 280–289, 1977. ISSN 0004-5411. http://dx.doi.org/10.1145/322003.322011.

[24] Kopacz T.: Programming in Windows Azure. *Presentation from the 4th MIC conference ,,Nowoczesne technologie bliżej nas"*, Poznan, Poland http://mic.psnc.pl/download/conf10/MIC_NTBN10_Azure.pdf, 2010.

[25] Król D., Wrzeszcz M., Kryza B., Dutka Ł., Kitowski J.: Massively Scalable Platform for Data Farming Supporting Heterogeneous Infrastructure. In: *The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization, IARIA Cloud Computing 2013*, pp. 144–149. Valencia, Spain, 2013.

[26] Marco J., Campos I., Coterillo I.e.a.: The interactive European grid: Project objectives and achievements. *Computing and Informatics*, 27(2): 161–171, 2008. ISSN 1335-9150.

[27] Mell P., Grance T.: *The NIST Definition of Cloud Computing - Recommendations of National Institute of Standards and Technology.* Special Publication 800-145, NIST US DOC, Sept. 2011 http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[28] Netto M. A., Buyya R.: Coordinated rescheduling of Bag-of-Tasks for executions on multiple resource providers. *Concurr. Comput. : Pract. Exper.*, 24(12): 1362–1376, 2012. ISSN 1532-0626. http://dx.doi.org/10.1002/cpe.1841.

[29] Oprescu A., Kielmann T.: Bag-of-Tasks Scheduling under Budget Constraints. In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 351–359. 2010. http://dx.doi.org/10.1109/CloudCom.2010.32.

[30] Pandey S., Karunamoorthy D., Buyya R.: *Workflow Engine for Clouds.* In: Cloud Computing: Principles and Paradigms, R. Buyya, J. Broberg, A. Goscinski (Eds.), pp. 321–344. John Wiley & Sons, Inc., New York, USA, 2011. ISBN 9780470940105. http://dx.doi.org/10.1002/9780470940105.ch12.

[31] Schubert L., Assel M., Kipp A., Wesner S.: *Resource Cloud Mashups.* In: Cloud Computing: Principles and Paradigms, R. Buyya, J. Broberg, A. Goscinski (Eds.), pp. 533–548. John Wiley & Sons, Inc., New York, USA, 2011. ISBN 9780470940105. http://dx.doi.org/10.1002/9780470940105.ch21.

[32] Senger H., da Silva F. A. B., Miranda Filho L. J.: Influence of Communication Models on the Scalability of Master-Slave Platforms Running Bag-of-Tasks Applications. In: *Proceedings of the 2010 11th Symposium on Computing Systems*, WSCAD-SCC '10, pp. 25–32. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-0-7695-4274-4. http://dx.doi.org/10.1109/WSCAD-SCC.2010.27.

[33] Suchta A., Marszałek K., Smoktunowicz U.: *Little rain from great cloud (Z dużej chmury mały deszcz)*. Article on CRN website: http://www.crn.pl/artykuly/biznes/2011/09/z-duzej-chmury-maly-deszcz, 2011.

[34] Voorsluys W., Broberg J., Buyya R.: *Introduction to Cloud Computing*. In: Cloud Computing: Principles and Paradigms, R. Buyya, J. Broberg, A. Goscinski (Eds.), pp. 1–41. John Wiley & Sons, Inc., New York, USA, 2011. ISBN 9780470940105. http://dx.doi.org/10.1002/9780470940105.ch1.

[35] Wismüller R., Bubak M., Funika W.e.a.: Support for User-Defined Metrics in the On-line Performance Analysis Tool G-PM. In: *Grid Computing – Second European AcrossGrids Conference AxGrids2004*, M.D. Dikaiakos, ed., *Lecture Notes in Computer Science*, vol. 3165, pp. 159–168. Springer-Verlag, Nicosia, Cyprus, 2004.

Affiliations

Włodzimierz Funika
AGH University of Science and Technology, ACC CYFRONET AGH, Krakow, Poland, funika@agh.edu.pl

Kamil Mazurek
AGH University of Science and Technology, Krakow, Poland, contact@kamilmazurek.pl

Wojciech Kruczkowski
AGH University of Science and Technology, Krakow, Poland, kruczkowski.wojciech@gmail.com

Received: 20.06.2013
Revised: 10.10.2013
Accepted: 20.12.2013