CLOUD MY TASK – A PEER-TO-PEER DISTRIBUTED PYTHON SCRIPT EXECUTION SERVICE

Eng. Daniel RIZEA[1], Eng. Daniela ENE[2], Eng. Rafaela VOICULESCU[3], Lecturer Eng. Ph.D. Mugurel Ionut ANDREICA[4]

Abstract

In this paper we present „Cloud My Task", a distributed Python script execution service. The service can receive Python scripts from the clients and execute them on its instances, according to their load. The service is fault-tolerant : the failure of any single instance will not disrupt the service availability. The instances are connected with each other in a peer-to-peer tree-like topology. Experimental results show that the service scales well and is able to balance the load across the available service instances.

Keywords: *Cloud, task scheduling, distributed service, load balancing, P2P, topology.*

1. INTRODUCTION

In this paper we intoduce „*Cloud My Task"*, a distributed Python script execution service implemented in the Java programming language. The service can distribute CPU-intensive Python scripts in a cloud environment balancing the computations and returning the results to clients as soon as they are available.

The motivation for focusing on Python scripts/programs is that Python is an easy and intuitive language that can be learnt very fast so that non-programmers can develop scripts in a matter of minutes. Scripts in general can run for long periods of time and may need CPU resources which are not available locally. Running these scripts in a cloud environment like „Cloud My Task" will alleviate the need for available local resources. „Cloud My Task" will also store the results of running the script for a period of time, so that these results may be retrieved by the users later.

The rest of this paper is structured as follows. In Section 2 we present the main features of the „Cloud My Task" service. In Section 3 we discuss service implementation details. In Section 4 we present the architectural design of the service. In Section 5 we discuss the TPSI topology. In Section 6 we present experimental results. In Section 7 we discuss related work and in Section 8 we conclude and discuss future work.

2. FEATURES OF THE „CLOUD MY TASK" SERVICE

The „Cloud My Task" service can run CPU-intensive Python scripts in a cloud environment with a high uptime. The machines composing the service are geographically close, usually in the same cluster or local area network. This way, both point-to-point and IP multicast communication between machines can be employed. The service instances are interconnected into a peer-to-peer topology, where each instance has a limited number of neighbors. Moreover, the service is fault-tolerant - the failure of a machine will not lead to the disruption of the service.

In order to use the „Cloud My Task" service, a client must communicate with an instance *IA* of the service (running on one of the machines). We will not address the service discovery

[1] Politehnica University of Bucharest, Computer Science Department, e-mail: danielrizea27@gmail.com
[2] Politehnica University of Bucharest, Computer Science Department, e-mail: daniela.a.ene@gmail.com
[3] Politehnica University of Bucharest, Computer Science Department, e-mail: rafaela.voiculescu@gmail.com
[4] Politehnica University of Bucharest, Computer Science Department, e-mail: mugurelionut@gmail.com

problem in this paper (i.e. we will not focus on mechanisms which allow the client to find an instance of the service to communicate with). The client sends its Python script to the service instance in order to be run. If *IA* is not too loaded, it will add the script to its processing queue and the script will be run as soon as possible. If the load on the instance *IA* is too high and the instance cannot process the demand right away, it will try to find another instance of the service with a load below the critical threshold. If such an instance *IB* is found, then the task will be forwarded to that instance and when the task will be done the result will be returned to *IA*. This way, the client only ever needs to communicate with *IA*, although its script may be run on a different service instance. At a certain point when all the instances are highly loaded, a script will wait for the CPUs to be free (in a special queue). The instance *IA* will sense when this happens and will pass the waiting script to a less loaded instance of the service (or run the script itself if it becomes less loaded).

The output for a script will be stored for a certain period of time. The client will get a unique id associated with that script and will be able to query the service for the result of the script.

3. SERVICE IMPLEMENTATION DETAILS

The service implementation will be analysed based on the interaction between the service clients and service instances. We will have client-task processing service instance (TPSI) and TPSI-TPSI interactions. We will first describe the main components of our services and then describe the interactions between them.

3.1. Service Components

3.1.1. Clients

A public interface will be offered to client application in order to be able to use our service. They will mostly use one method:

public Integer submitJob(ClientID, QoS type, Script) { ... }

The command will return the script job ID in the system. Clients will then be able to query for the status of the script they have submitted using the script job ID obtained from submitting the script, using the method below:

public String getScriptJobStatus(ScriptJobID) { ... }

This method will return the script result or will return the status of the current script. (Pending, Running, Waiting, etc.).

3.1.2. Task Processing Service Instance (TPSI)

This is the main part of our service. The TPSIs execute Python scripts and returns their results to client application. A TPSI communicates with client applications to get script jobs, it communicates with other TPSI instances to pass tasks so that load balancing is achieved in the system and it also communicates with the Central Supervisory Unit to query and find out if users are in a banned list.

3.1.3. Central Supervisory Unit

This is a supervisory unit that will periodically collect system information and overall state. For now it will hold a client blacklist that will prevent clients from achieving a DoS attack on the distributed system. Each time a TPSI senses the possibility of an attack it will

reject the client's job requests and call *addToBlackListRequest* to the Central Supervisory Unit.

3.2. Interactions between components

We will present the main types of **requests** based on the components that are interacting. The requests are broken down into smaller steps. Each of these steps composes a stage of the request processing pipeline and will be executed by a thread from a pool of threads (either created on demand or statically created in the beginning). The results from a previous stage will be submitted in the job queue of the next stage along with other context information. Based on the **3 possible interactions** we will have **3 main** requests types available.

3.2.1. Client job request

This type of request is submitted from a client to a Task Processing Service Instance (TPSI). This request involves the submission of a Python script along with other identifiers. The processing stages of this type of request are shown below (and also presented in Fig. 1):

1. Receive the Request (get the request and the Python script).
2. Decode the Request (decide the request type and what logical flow to follow).
3. Submit request to Central Supervisory Unit to check if client is banned from the service and wait for response.
4. If the client is banned then return a response that it is banned from the service, else continue to step 5.
5. Decide if the local TPSI is able to serve the client's request or decide to pass the request to a neighbor (based on the load and topology information received from the other TPSIs ; basically, another TPSI from the whole system with load below threshold is chosen randomly among all the TPSIs with loads below threshold).
6. If the request is executed on the local machine then execute it, otherwise wait for it to be executed on another machine.
7. Send the response back to client.

3.2.2. TPSI - TPSI job request

This type of request occurs when the TPSI that received a client job is full and tries to find a "free" TPSI to submit the job to it (a TPSI is "free" if its CPU load is below a predefined threshold). The submitter TPSI acts like a client for the TPSI that receives the request. The only difference is that the TPSI that receives the request doesn't verify the banned status with the Central Supervisory Unit (the submitted TPSI is trusted). The processing stages of this type of request are:

1. Receive the Request (get the request and the Python script).
2. Decode the Request (decide the request type and what logical flow to follow).
3. Decide if the local TPSI is able to serve the client's request or decide to pass the request to a neighbor.
4. If the job is executed on the local machine then execute it, otherwise wait for it to be executed on another machine.
5. Send the response back to client.

3.2.3. TPSI - TPSI IP Multicast communication

Due to their geographical proximity, all the TPSIs are part of the same IP Multicast group. Periodically, each TPSI broadcasts information regarding its load and regarding its neighbors (topology). This information is used by the TPSIs in order to decide where to send a client job when the current TPSI is too loaded.

The Central Supervisory Unit is part of the same IP Multicast group, so it is involved in all

the IP Multicast communication.

3.2.4. TPSI - Central Supervisory Unit interaction

This type of TPSI – Central Supervisory Unit (CSU) interaction occurs when the TPSI wants to verify that the user is not in a banned list. A user can enter a banned list if it attempts to create a DOS attack by submitting a large number of CPU intensive scripts in a relative short period of time. To avoid such a behavior the TPSI will notice the large number of requests and will deny them and send a request to add the user to the banned list for a certain period of time.

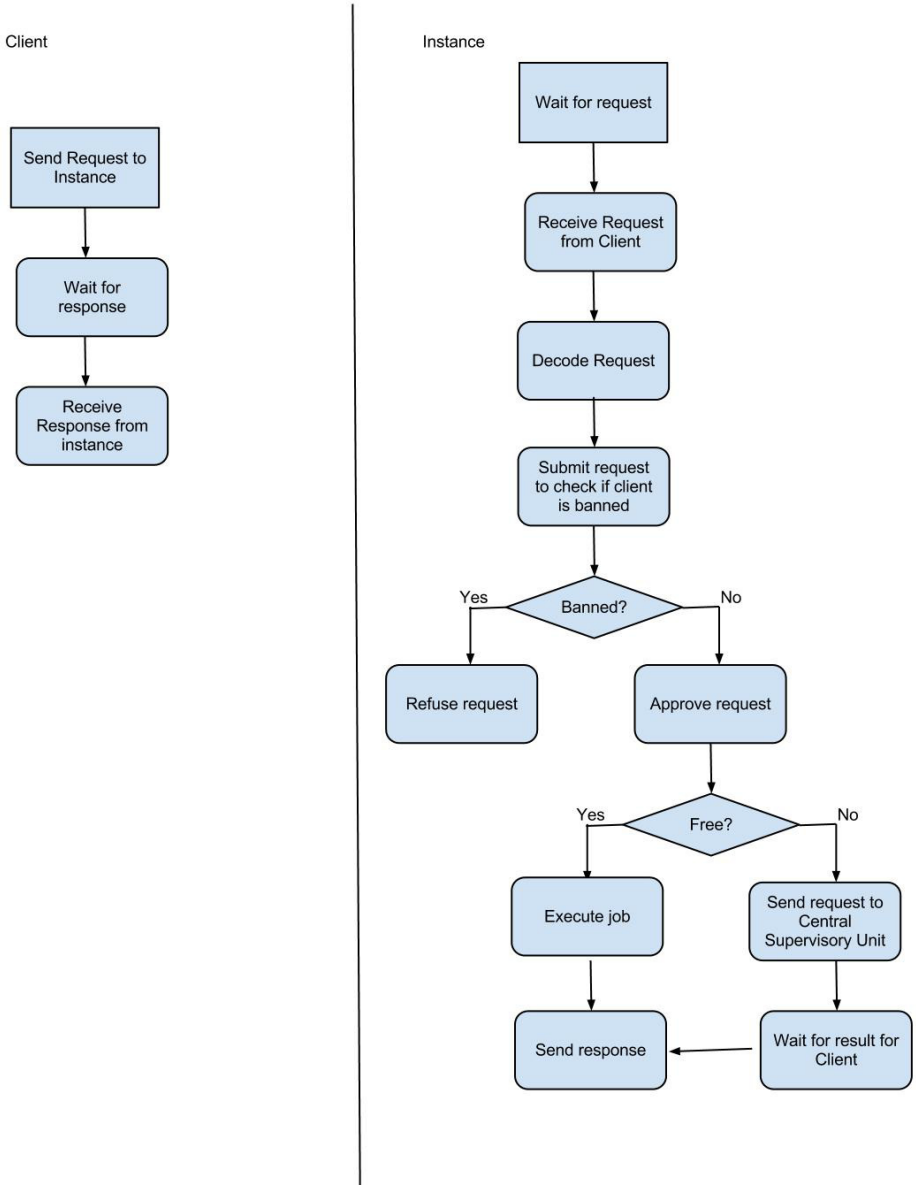


Figure 1. Client-TPSI interaction.

4. ARCHITECTURAL DESIGN

The architectural design of the service's components is modular. The architecture is separated into different layers with public interfaces in between the layers. We used a concept similar with the OSI Stack, each 2 neighboring layers being able to communicate through their service interfaces. The advantages of such a design is that one can completely change the

inner logic of a module and use the same interfaces and no other modules are affected by the change. The main modules of our architecture are presented in Fig. 2.

Point-to-point communication between clients and TPSIs or between two TPSIs is configurable and can be performed by using either UDP or TCP sockets, in blocking or non-blocking (Java NIO) mode. Fig. 3 presents a different view of the architectural design, focused more on the interactions between the service components.

5. SERVICE TOPOLOGY

In order to ensure service scalability a full mesh topology between the TSPIs is not the answer. Instead we use a peer-to-peer tree-like topology. Each instance has a number of neighbor instances. In order to communicate with non-neighboring instances, other instances will be used for intermediate routing of the messages (requests and responses). Currently, the neighbors of each instance are statically configured.

In order to avoid the Central Supervisory Unit being a single point of failure we implemented a response timer on the TPSI. If no response is received the request will advance to the next processing stage.

The generic service topology is presented in Fig. 4.

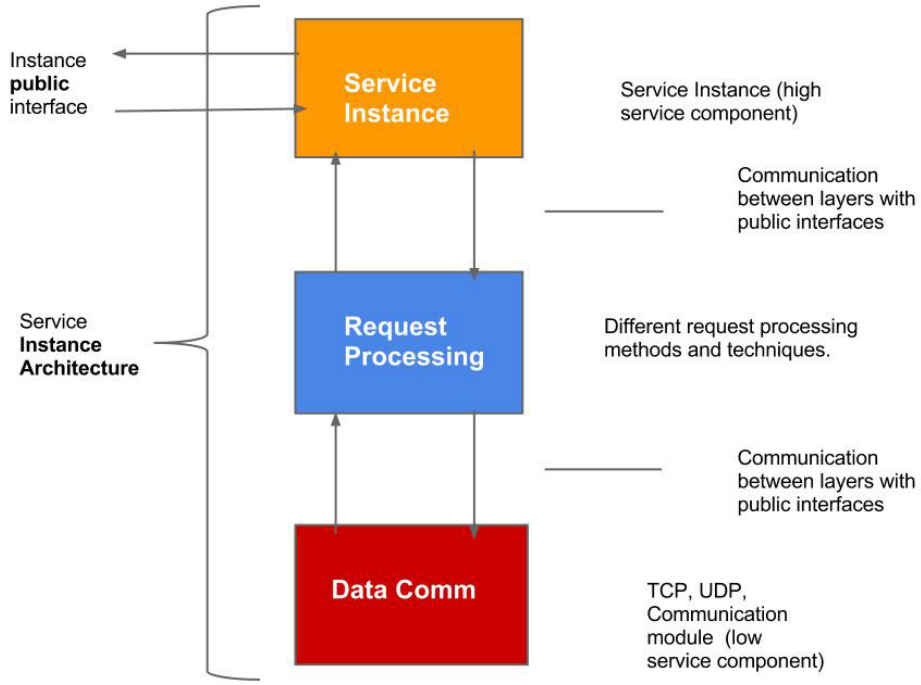


Figure 2. Main modules of the service architecture.

6. EXPERIMENTAL RESULTS

In order to compute relevant metrics during testing, we instrumented the TPSIs with the ApMon library [3] which is capable of sending monitoring data to the MonALISA distributed monitoring framework [2]. The monitoring metrics were explicitly computed by the service instances. We used the MonALISA framework in order to visualize the relevant experimental evaluation metrics.

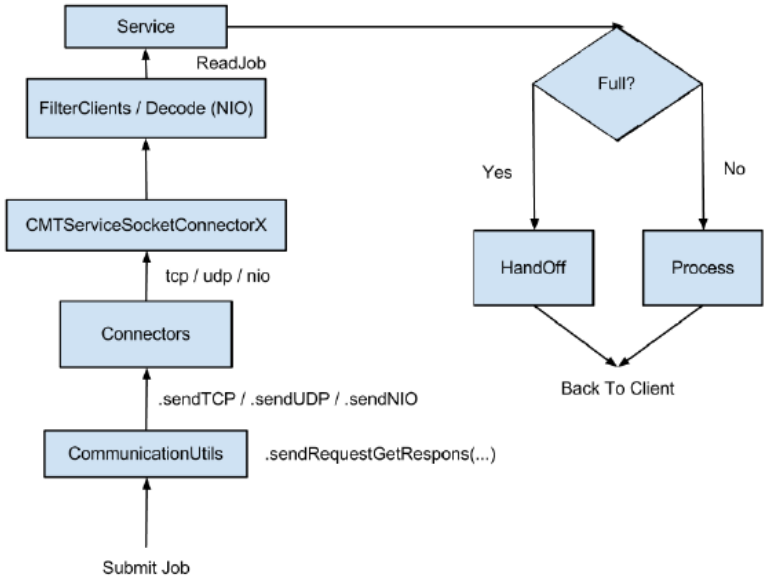


Figure 3. Architectural design and interactions between service components.

We used 5 service instances (TPSI) in our test scenarios, running on 5 different machines in a local area network, and 4 clients. Each client submitted to the service copies of the same CPU-intensive Python script having a 5 seconds duration. To be more precise, each client ran in a loop. At each iteration, each client submitted a certain number of copies of the 5-seconds Python script and then waited for 5 seconds before starting the next iteration. All the clients behaved identically in all the tests.

We ran 3 types of tests. The first type was labeled the "normal" test. Each client submitted at most 6 jobs per iteration. The second type was labeled the "burst" test. Each client alternatively submitted either at most 4 jobs per iteration or at most 8 jobs per iteration. The third type of test was labeled "full-capacity" test. In this case each client submitted up to 8 jobs per iteration. The exact number of jobs submitted by each client was chosen randomly between 1 and the upper bound mentioned above. Within each test, each TPSI is capable of executing at most 4 Python scripts in parallel (it has a thread pool consisting of 4 threads which is dedicated to executing the scripts). Its load is computed as a percentage from 0 to 100, depending on the number of scripts it is currently running (0% load for 0 scripts and 100% load for 4 scripts). For each test we computed the variation in time of the load of the TPSIs, as well as the response time perceived by clients. The response time is equal to the time difference between the moment when the job is submitted and the moment when the result is received. Note that the response time can never be lower than 5 seconds, as that's the execution duration of the script. In terms of communication we considered both blocking and non-blocking TCP sockets. Fig. 5-10 show the metrics mentioned above for each type of test.

7. RELATED WORK

A distributed peer-to-peer processing system (OSIRIS) was presented in [1]. Load balancing mechanisms for task execution in heterogeneous peer-to-peer systems based on mobile agents were proposed in [4].

The case of splittable jobs was considered in [5]. There the authors consider the problem of online scheduling splittable tasks on multiple parallel machines, with applications to a peer-to-peer setting. Ant-colony and particle swarm optimization techniques for achieving improved load balancing in Grid environments were proposed in [6], while genetic algorithms-based scheduling approaches were presented in [8]. The problem of re-scheduling

tasks in distributed systems (Grid environments in particular) was addressed in [7].

In [10] a system which is capable of using public computing and storage resources for running jobs and solving tasks is presented. By comparison, our service is run on dedicated machines which are owned by the service provider (and usually located in a cluster-like environment).

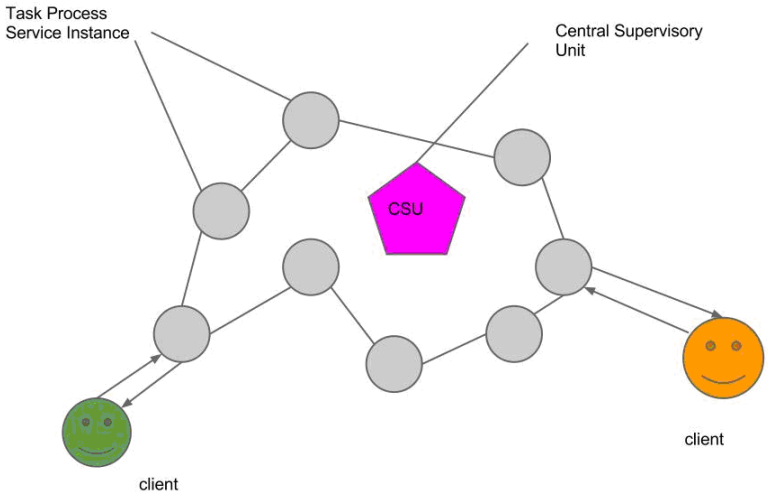
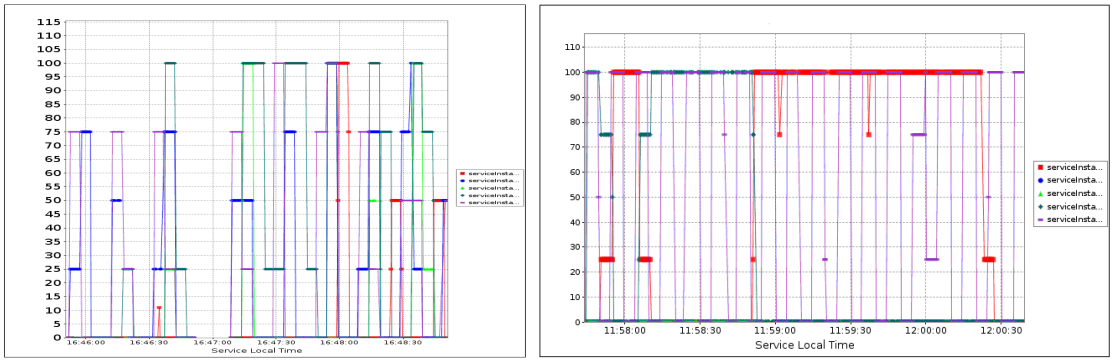


Figure 4. Service topology.



(a) **(b)**

Figure 5. „Normal" test – TPSI load variation in time: a) Blocking TCP; b) Non-blocking TCP (Java NIO).

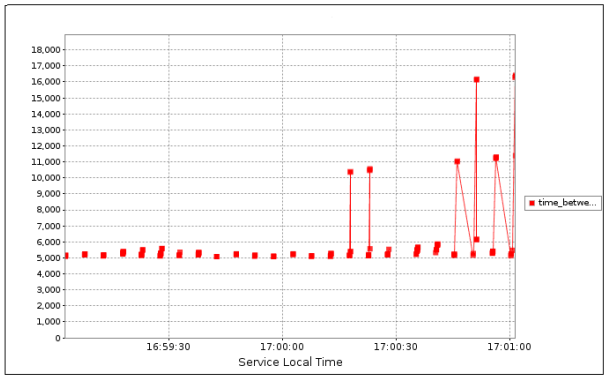
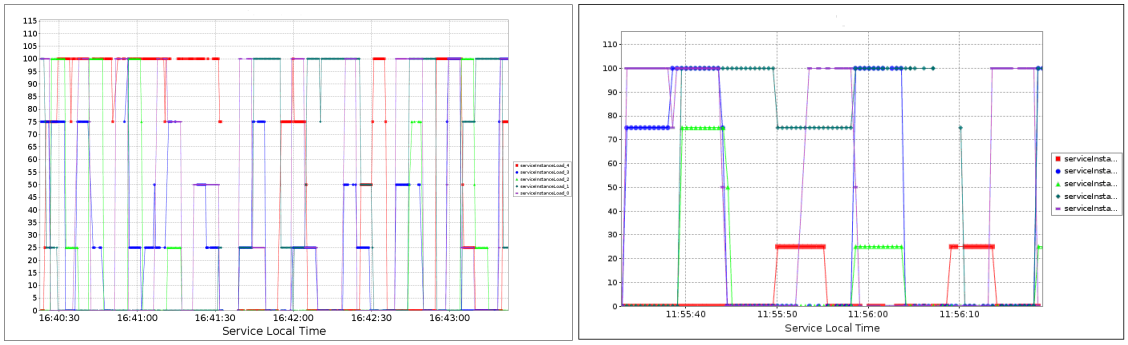
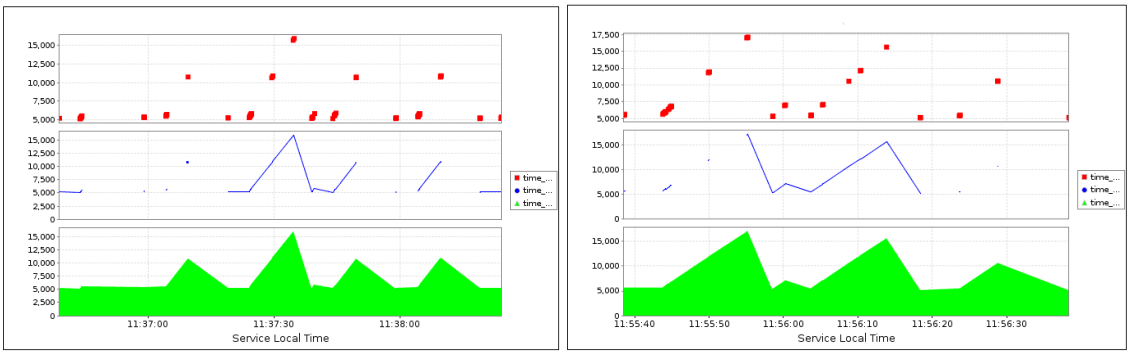


Figure 6. „Normal" test – Response time perceived by the clients (sec) using blocking TCP sockets. Similar results were obtained with non-blocking TCP sockets.



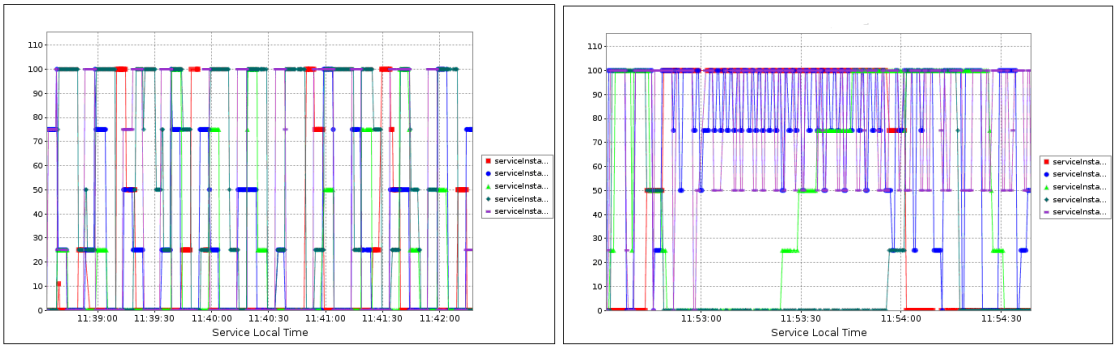
(a) **(b)**

Figure 7. „Burst" test – TPSI load variation in time: a) Blocking TCP; b) Non-blocking TCP (Java NIO).



(a) **(b)**

Figure 8. „Burst" test – Response time perceived by the clients (sec): a) Blocking TCP; b) Non-blocking TCP (Java NIO).

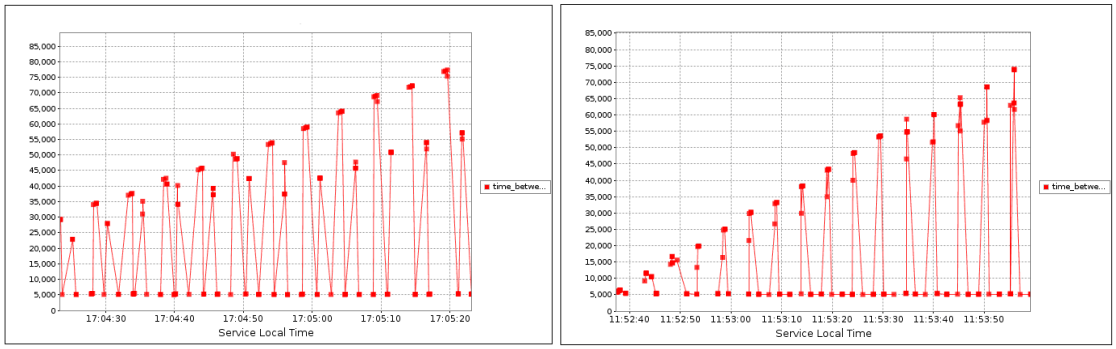


(a) **(b)**

Figure 9. „Full-capacity" test – TPSI load variation in time): a) Blocking TCP; b) Non-blocking TCP (Java NIO).

8. CONCLUSIONS AND FUTURE WORK

In this paper we presented „Cloud My Task", a distributed service for executing Python scripts. The service scales well to multiple instances and is capable of balancing the (CPU) load across all the currently available instances. The service is also fault-tolerant, meaning that the failure of a single instance will not disrupt the availability of the service. The service instances are interconnected into a peer-to-peer tree-like topology. Experimental results showed that the service works as intended, i.e. scales well and balances the load properly.



(a) **(b)**

Figure 10. „Full-capacity" test – Response time perceived by the clients (sec)): a) Blocking TCP; b) Non-blocking TCP (Java NIO).

However, there still are a number of improvements which could make the „Cloud My Task" service better suited to the needs of its clients. First of all, the topology management of the service instances could be improved and a dynamic method, like the ones used for handling DHT topologies, could be employed [9]. In order to increase the service's scalability, we intend to decrease its dependency on a single Central Supervisory Unit. We will break the TPSIs into groups and assign to each group a local Central Supervisory Unit. This will reduce the workload on the Central Supervisory Unit and will result in a more flexible and scalable service.

9. REFERENCES

[1] C. Schuler, R. Weber, H. Schuldt, H.-J. Schek, „Peer–to–Peer Process Execution with Osiris", Lecture Notes in Computer Science, Vol. 2910, pp. 483-498, 2003.
[2] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, C. Stratan, „MonALISA: An Agent based, Dynamic Service System to Monitor, Control and Optimize Distributed Systems", Computer Physics Communications, Vol. 180, No. 12, pp. 2472-2498, 2009.
[3] ApMon. http://monalisa.caltech.edu/monalisa__Service_Applications__ApMon.html
[4] N. Nehra, R. B. Patel, V. K. Bhat, „Load Balancing in Heterogeneous P2P Systems using Mobile Agents", World Academy of Science, Engineering and Technology, vol. 20, pp. 1014-1019, 2008.
[5] L. Epstein, R. van Stee, „Online Scheduling of Splittable Tasks in Peer-To-Peer Networks", Lecture Notes in Computer Science, Vol. 3111, pp. 408-419, 2004.
[6] S. A. Ludwig, A. Moallem, „Swarm Intelligence Approaches for Distributed Load Balancing on the Grid", Journal of Grid Computing, Vol. 9, No. 3, pp. 279-301, 2011.
[7] F. Pop, C. Dobre, C. Negru, V. Cristea, „Re-scheduling Service for Distributed Systems", Advances in Intelligent Systems and Computing, Vol. 187, pp. 423-437, 2013.
[8] G. V. Iordache, M. S. Boboila, F. Pop, C. Stratan, V. Cristea, „A Decentralized Strategy for Genetic Scheduling in Heterogeneous Environments", Multiagent and Grid Systems, Vol. 3, No. 4, pp. 355-367, 2007.
[9] Y. M. Zhang, X. Lu, D. Li, „Survey of DHT Topology Construction Techniques in Virtual Computing Environments", Science China Information Sciences, Vol. 54, No. 11, pp. 2221-2235, 2011.
[10] D. P. Anderson, „BOINC: A System for Public-Resource Computing and Storage", Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004.