Performance testing of distributed computational resources in the software development phase

Jozef Cernak[*], **Eva Cernakova and Marek Kocan**
P. J. Safarik University in Kosice, Kosice, Slovak Republic
E-mail: `jcernak@upjs.sk`

A grid software harmonization is possible through adoption of standards i.e. common protocols and interfaces. In the development phase of standard implementation, the performance testing of grid subsystems can detect hidden software issues which are not detectable using other testing procedures. A simple software solution was proposed which consists of a communication layer, resource consumption agents hosted in computational resources (clients or servers), a database of the performance results and a web interface to visualize the results. Communication between agents, monitoring the resources and main control Python script (supervisor) is possible through the communication layer based on the secure XML-RPC protocol. The resource monitoring agent is a key element of performance testing which provides information about all monitored processes including their child processes. The agent is a simple Python script based on the Python *psutil* library. The second agent, provided after the resource monitored phase, records data from the resources in the central MySQL database. The results can be queried and visualized using a web interface. The database and data visualization scripts could be considered for a service thus the testers do not need install them to run own tests.

EGI Community Forum 2012 / EMI Second Technical Conference,
26-30 March, 2012
Munich, Germany

[*]Speaker.

1. Introduction

Performance testing in the development phase of a software production can provide early feedback about potential performance issues. The developers can propose and implement effective solutions to solve the performance issues before products are widely deployed [1, 2]. The EMI testing policy [3] covers performance testing activities during certification i.e. before public software deployment.

The performance testing before wide deployment has specific needs. In some cases, for example memory leaks, more detailed information is needed not only about running daemons and processes (CPU times, memory usage, open ports and etc.) but also we need to know resource consumption of their child processes. The requirements on tool are: (i) an installation of the tool should be simple and quick with minimal maintenance, (ii) the tool should be flexible to easily realize new test cases. After a short review of available open tools we proposed to design a simple proprietory solution [5] which should meet all our required criteria i.e. easy set up and flexibility in usage with minimal maintenance.

2. Basic components of the performance tool

The performance tool consists of two Python scripts which behave as agents, central DB with a web interface to visualize the results, and communication layer Fig. 1.

One of the scripts, *monitor_process_tree* collects information about monitored processes and all their subprocesses. The script utilizes the Python *psutil* library [4] which is available for Linux and MS Windows. The second Python script *send_proc_tree* sends data from a monitored resource to the central DB. The activity of the scripts (agents) can be controlled manually, automatically or in a combination of manual and automatic control. In the manual mode a developer or tester manually starts or stops scripts *monitor_process_tree* and *send_proc_tree* on the resources. When monitoring of resources is stopped then script *send_proc_tree* sends the results to the central DB. This mode is the easiest way to obtain required performance parameters. Scripts are started:

```
monitor_process_tree.py  "list of process names"
send_proc_tree.py  "list of process names"
```

In the automatic mode we need to run main Python script which sets up required parameters and initiates activity of basic scripts *monitor_process_tree* and *send_proc_tree* i.e. agents running on the resources. The main script depends on the test case. A secure XLM-RPC protocol is used for communication between resources and the main script. This simple structure can cover complex tests including monitoring of performance of processes running on several distributed resources (clients and servers).

In many simple cases, a developer needs to install *psutil* library [4] and start two simple python scripts. For more complex tests, the developer can use the same approach or write own test script which synchronize events.

The central database and visualization scripts could be considered for service to store and visualize the results from independent developers i.e. the developer does not need to install and maintain database. The proposed scenario is slightly different from the one that is adopted in

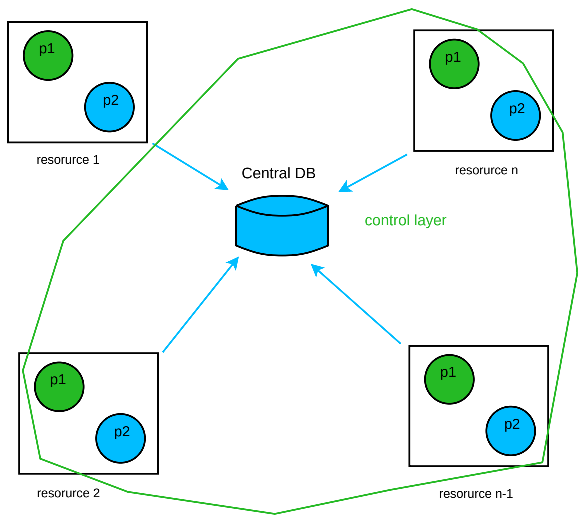


Figure 1: Main blocks of the performance test tool. Processes are p1 - *monitor_process_tree* and p2 - *send_proc_tree*

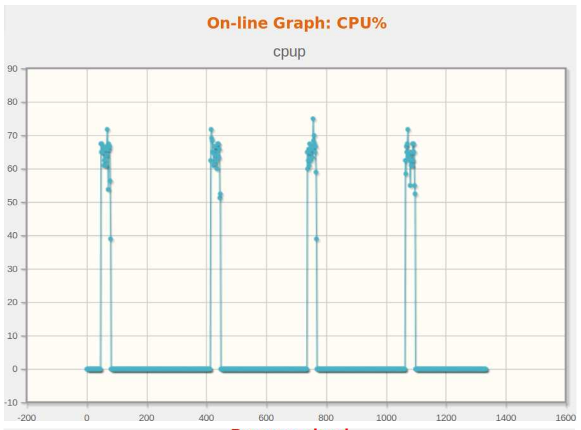
other monitoring system which are designed to monitor performance and reliability of servers and services during long period, for example Nagios.

3. Performance test cases

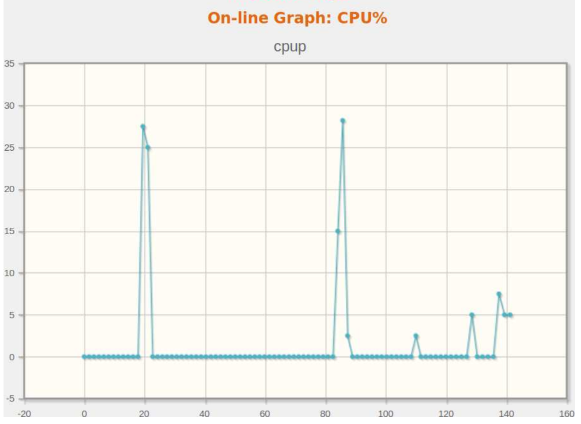
We separated performance testing in a few phases such as monitoring of processes, running of test cases, sending the results to DB, and visualization of the results. This approach allows the developer to run arbitrary own test without additional modifification what could be considered for advantage. The simplicity of usage is demonstrated by two simple test cases. The number of system parameters which we can assign to the process is limited by *psutil* library [4], however for many test cases the set of parameters is complete.

The first test case covers testing of client-server performance during submission of 1000 tasks which were sent from the client to the server (load testing). The server hosted scripts *monitor_process_tree.py* and *send_proc_tree.py* which were started and stopped manually. Daemons *arched* and *slapd* including all their subprocesses were monitored for several minutes with interval of 2 seconds. The results were stored in the central DB and are available via web interface [5].

The second case demonstrates parameter testing i.e. seeking the optimal parameter setting of the *slapd* daemon to increase performance of server. During testing no jobs were submitted and a low usage of CPU was expected (Fig. 2). Initially the server installed and configured with the 2nd release candidate of the EMI major release 2 (*EMI-2rc2*) showed that the *slapd* daemon consumed more CPU time than we expected (Fig. 2(a)). However, after a change of BDII and *slapd* settings in the newest release *EMI-2rc4* , the results showed less CPU usage see Fig. 2(b). Finally, we can track the performance parameters during development phase from a release to the next release.



(a)



(b)

Figure 2: Comparison of the CPU usage of the *slapd* process in the case when no jobs are submitted. The CPU usage for ARC CE for a software of version: (a) EMI-2rc2 and (b) EMI-2rc4. The x-axis is time in seconds and y-axis is CPU usage in %.

4. Conclusions

A prototype of performance tool based on Python library *psutil* has been developed and tested. The prototype meets all our initial requirements. We assume that the scripts (agents) could be easily integrated in similar performance tools, for example Nagios.

We plan to use the tool during the certification testing for the EMI-2 release as additional tool to Nagios probes which are currently deployed. However, more complex performance and profiling analysis of all ARC components will be realized after the EMI-2 release.

We have identified a need to define widely acceptable performance test cases and minimal performance parameters across grid software vendors. For example, a number of submitted jobs per minute or reliability of job submission process, which could enable potential users to evaluate a quality of distributed software solutions.

5. Acknowledgment

This work was done as part of the EMI project and partially funded by the European Commission under Grant Agreement INFSO-RI-261611.

References

[1] D5.4-1 RESOURCE CONSUMPTION PROFILE AND PERFORMANCE BENCHMARK STUDY OF THE EARLY PROTOTYPE RELEASE OF KNOWARC: `http://www.knowarc.eu/documents/Knowarc_D5.4-1_07.pdf`

[2] D5.4-1 RESOURCE CONSUMPTION PROFILE AND PERFORMANCE BENCHMARK STUDY OF THE FINAL RELEASE OF KNOWARC: `http://www.knowarc.eu/documents/Knowarc_D5.4-1_09.pdf`

[3] EMI testing policies: `https://twiki.cern.ch/twiki/bin/view/EMI/EmiSa2TestPolicy`

[4] psutil cross-platform process and system monitoring module for Python: `http://code.google.com/p/psutil/`

[5] ARC tools for revision, functional and performance testing (including DB of test results and tool to generate EMI test reports): `http://arc-emi.grid.upjs.sk/tests.php`

[6] ARC test coordination: `http://wiki.nordugrid.org/index.php/Testing`