Monitoring and analysing a Grid Middleware Node

Ramon Nou [#1], Ferran Julià [#2], David Carrera [#3], Kevin Hogan [*4], Jordi Caubet [+5], Jesús Labarta [*6], Jordi Torres [*7]

[#] *Computer Architecture Departmenr, Technical University of Catalonia*
Campus Nord, Jordi Girona 1-3, E08034 Barcelona, Spain
[1,2,3]`{ rnou, fjulia, dcarrera }@ac.upc.edu`

[*] *Barcelona Supercomputing Center(BSC)*
Jordi Girona, 29 - E08034 Barcelona, Spain
[4,5,6]`{kevin.hogan , jesus.labarta, jordi.torres }@bsc.es`

[+] *IBM Innovation Initiative (I3-BSC)*
Jordi Girona, 29 - E08034 Barcelona, Spain
[5]`jordi_caubet@es.ibm.com`

***Abstract*— Distributed Grid applications are becoming more and more popular as the use of complex grid middlewares becomes extensive, and more facilities are offered by these complex pieces of software. But as Grid middlewares grow and offer more advanced features, they become more complex and weighty, as well as hard to tune. As the performance of a distributed Grid applications can be strongly influenced by the operation of the underlying grid middleware, it becomes importan to study and analyze their behaviour and performance. In this paper we present the eDragon Monitoring Framework (eDMF), a set of tools for instrumentation and analysis of grid middleware, that provides an unique environment to study the performance of Grid applications. The eDMF is composed of a set of specialised monitoring tools as well as a flexible and powerful performance analysis platform. Additionally we also provide a practical application of the eDMF to the Globus Toolkit 4 (GT4), one of the most extended and popular Grid middlewares, showing how it helped us in the detection and resolution of several job management problems observed in the GT4 middleware.**

I. INTRODUCTION

Nowadays, the performance analysis of grid middleware [1], [2] is an important subject since it has an important impact on the global performance of grid applications. However, as the complexity of the grid middleware increases, the performance analysis tools must evolve and increase their effectivity and precision to keep being useful and to provide new insights on the performance of grid applications and middleware.

In this poster we present the eDragon Monitoring and Analysis Framework (eDMF), an instrumentation and analysis framework developed at Barcelona Supercomputing Center (BSC) that helps to study the performance of the core functionalities of a grid node, considering the grid application, the grid middleware and the underlaying levels as a whole. This novel feature allows an in-depth highly detailed performance analysis of all the components of a grid environment putting all them in relation to each other as well as in relation to the execution platform, making it possible to detect performance problems as well as their root causes, and provides an extremely detailed insight into the performance issues involved in the efficient execution of grid applications.

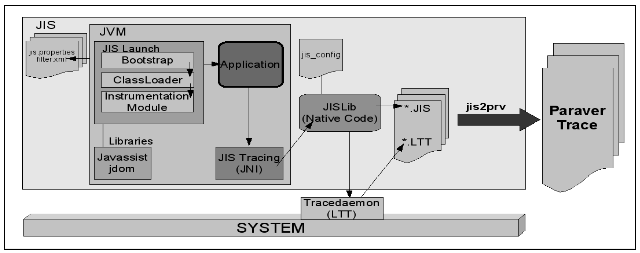

Fig. 1. eDMF structure

II. EDRAGON MONITORING FRAMEWORK

The eDragon Monitoring Framework (eDMF) was developed to monitor and measure the performance of Java applications, track the underlying system running them and display all of this data in an easily digestable format for analysis. The idea is to first use eDMF to spot any inefficiencies or issues of contention on the system as a whole and then to later use it to test different settings/changes which could solve the issue at hand. Solutions could involve reprogramming routines, changing configuration files for the application, changing the system configuration amongst many others.

This approach is in stark contrast to other Java profilers/performance monitors since they generally focus on the application only and view things from that perspective. While these profilers are useful for the development cycle of applications, they often do not have much usefulness during the deployment and running of them in the real world. In complex application servers, where external processes may be sapping resources or causing bottlenecks, this view is too simplistic and ultimately inadequate to determine the applications performance correctly.

The eDMF consists of three separate parts that all work together to achieve our aims. The first part is called the Java Instrumentation Suite (JIS) [3] and contains some tools to trace running Java applications on a JVM as well as parsing tools for merging different tracefiles. The Linux Trace Toolkit (LTT) [4] is a well known system developed to trace the processes running on a Linux OS. Paraver [5] is a flexible parallel program visualisation and analysis tool and was selected as the best way to visualise the final traces. See figure 1 for a visual description of how they work together. In tandem with the goals of the eDMF, it is important for it to have a low overhead and allow the monitored application to be run in a normal environment (as opposed to in a development or debug mode). This placed certain constraints on how the tools could be developed and restricted the use of many of the standard features supplied with most modern JVMs to measure performance.

As can be seen from the diagram in figure 1, at the end of the tracing run there are plenty of files left in the output directory. The traces generated by JIS have the file description .jis and there is one created for each thread of the application. As mentioned in the paragraph above there is a ltt binary file for each CPU and the file description is .ltt. To combine everything together into a single trace, a tool called jis2prv merges them for use with Paraver.

Viewing and analysing the final trace is left for Paraver. This tool can be used to great effect to provide detailed quantitative analysis of an applications performance. It is a flexible system that has an enormous amount of options which can be used to obtain specific views on the trace.

III. Problem analysis

When we did some tests over a GT4 installation, trying the maximum number of jobs that we can be submit, we found that the amount of finished jobs decreases when we increase the number of jobs submitted. Another problem we recognised during these tests is the increase in the response time on the client side. To figure out what is giving us these symptoms we should analyse it using an all-level monitoring framework, as the problem could be at system level or at application (under a JVM) level.

The GT4 container uses two sets of threads. The first, ServiceDispatcher (only one thread), works by attending to client connections and sending the requests to a RequestQueue. The second set, called Service threads, receive the request from the RequestQueue and process it. The number of these threads varies dynamically depending on the server load. The processing involves security and SOAP processing too which is made using the axis classes.

The Gram service enables the remote execution of the job. In our case we don't use the RFT (Remote File Transfer) as we don't transfer any files, so we can divide the service in two abstract parts: the one managing the system at Java level, At last the execution of the job is prepared, interacting with the local scheduler. In our case, for simplicity we use the fork scheduler provided with GT4 software.

In our test we have some jobs that overload the CPU, using a for-loop that creates 5 seconds of 100% CPU load. In this case we execute 128 of these jobs in parallel. This benchmark overloads the CPU at system level, so it can produce the effect of reducing performance or adding inefficiencies at the middleware level, losing jobs in some test cases (more than 150 submitted jobs). Globus was able to execute 128 jobs in it's original version but was only able to execute a màximum of 158 jobs, by the other hand the modified version was able to execute a màximum of 238.Another good improvement was the increased number of jobs executed (and decreased response time).Globus continues trying to execute all the jobs as soon as possible (this is a desirable behaviour on a sleep-job type, but not on a CPU-intensive job). This behaviour produces several negative effects: Globus middleware locks up and response time increases. The jobs execution time lasts nearly 52 seconds on average (for a 5 seconds job). We got this information by using Paraver to analyse the last trace.

IV. Resource management Analysis

We should test a simple proposal that provides a way to limit the number of jobs that can execute on the system and increase its priority in order to decrease the number of context switching over this process. If we reduce this number, we should improve the servers performance allowing it to spend more time doing useful things (providing more CPU-affinity, too). We will analyse this proposal using paraver in order to propose or fine tune this first approach, and see if we can obtain interesting results, by viewing and analysing the system as a whole.

V. Conclusions

We have presented a framework that enables the monitoring and profiling of programs that involve different system/application layers, As far as we know there is no other tool capable of showing and managing the three levels (Application, JVM layer and system view) at once. A monitoring framework can help us to see problems where there doesn't seem to be any. Without the eDMF this could be an impossible task. Improvements on this framework are part of our future work and also some self-managing policies involving Grid Middleware.

Acknowledgement

This work is supported by the Ministry of Science and Technology of Spain and the EU under contract TIN2004-07739-C02-01.

References

[1] B. Sotomayor and L. Childers, *Globus Toolkit 4 : Programming Java Services*, M. Kaufmann, Ed., 2005.

[2] M. Romberg, "The unicore grid infrastructure," http://www.unicore.org, 2002.

[3] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, and J. Labarta, "Complete instrumentation requirements for performance analysis of web based technologies," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, USA*, March 2003.

[4] K. Yaghmour and M. R. Dagenais, "Measuring and characterizing system behavior using kernel-level event logging," *Usenix Annual Technical Conference, San Diego, CA*, June 2000.

[5] G. Jost, H. Jin, J. Labarta, J. Gimenez, and J. Caubet, "Performance analysis of multilevel parallel applications on shared memory architectures," 2003.