Monitoring and Analysis Framework for Grid Middleware

Ramon Nou[1], Ferran Julià[1], David Carrera[1], Kevin Hogan[2],
Jordi Caubet[3], Jesús Labarta[2] and Jordi Torres[2]

[1] Computer Architecture Department
Technical University Of Catalonia
Barcelona, Spain
{rnou, fjulia, dcarrera}@ac.upc.edu

[2]Barcelona Supercomputing Center
Centro Nacional de Supercomputación
Barcelona, Spain
{kevin.hogan, jesus.labarta, jordi.torres}@bsc.es

[3]IBM Innovation Initiative (I3-BSC)
Centro Nacional de Supercomputación
Barcelona, Spain
jordi.caubet@es.ibm.com

Abstract

As the use of complex grid middleware becomes widespread and more facilites are offered by these pieces of software, distributed Grid applications are becoming more and more popular. But as Grid middleware grows in size and offers more advanced features, they become more complex and heavier, as well as harder to tune. Since the performance of a distributed Grid application can be strongly influenced by the operation of the underlying grid middleware, it becomes of extreme importance to study and analyse its behaviour and performance.

In this paper we present the eDragon Monitoring Framework (eDMF), a set of tools that can be used for the instrumentation and analysis of grid middleware, and which provides a unique environment to study the performance of Grid applications. The eDMF is composed of a set of specialised monitoring tools as well as by 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 middleware, showing how it helped us in the detection and resolution of several job management problems observed in the GT4 middleware.

1 Introduction

Grid technologies have enabled the clustering of a wide variety of geographically distributed resources and services which can be used as a single execution endpoint, accessible over a network. Making network-available resources is achieved by using a set of functionalities and technologies, not provided by grid applications themselves, but by the grid middleware, a set of extremely complex pieces of software, commonly found as the basis of most grid technologies [9, 10].

Nowadays, the performance analysis of grid middleware 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 effectiveness and precision so as to continue being useful and to provide new insights on the performance of grid applications and middleware. The performance of the grid middleware can be studied from different perspectives (such as job scheduling and distributed resource management). For the scope of this article we will focus on the performance of a single grid node. Although there are other tools (see section 2 for more details), we are not only considering the performance issues related to the grid middleware but putting it in relation with the performance of the whole system (OS, execution platform, virtualization environment...).

In this paper, 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 grid middleware, taking into consideration the grid application, the grid middleware and the underlying 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. It makes 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.

As a testbed for the extensive capabilities of the eDMF framework, it was put to work on measuring the performance of the Globus Toolkit 4 (GT4) [10], one of the most promising and extended grid middleware systems in the market. A global view as well as a detailed statistical analysis of the execution was achieved.

The rest of the paper is structured as follows: in Section 2 we present a survey of available tools for monitoring,

15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)
0-7695-2784-1/07 $20.00 © 2007 IEEE

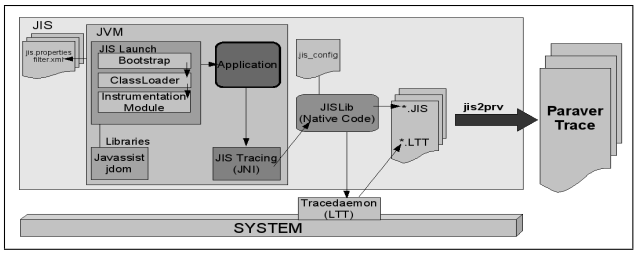

Figure 1. eDMF structure

in Section 3 we analyse the monitoring framework, in Section 4 we present some symptoms that we observed and how we analysed them with the framework. This analysis shows one possible solution, which we test in Section 5. Finally the conclusions are shown in Section 6.

2 Java Profiling Tools

There are currently a vast and varied amount of tools available for analysing Java applications. For our purpose, low overheads were a very high priority and since we needed to be able to extract full working models in the context of the whole system, we could not rely on sample based profilers. After careful consideration, we chose to use our own monitoring framework, eDMF, since it fullfilled the requirements completely. Highlights of some of the other applications we considered are listed below (an extended version can be found at [8]):

hprof is a standard JVMPI/JVMTI based agent supplied with Sun's JVMs for profiling but has high overheads. *Java Interactive Profiler (JIP)* is a profiler using aspects written for Java 5.0. It's XML output can be heavy for large traces. *Sun's JFluid JVM* is a modified JVM and may produce results at odds with a realistic deployment. *BEA JRockit 5.0* is a JVM which is reportedly 20% faster than Sun's but again, like any modified JVM, may not have much bearing on reality.

Java Performance Monitoring Toolkit (JPMT) [6],is a system wide tracing toolkit making some use of LTT, although its not currently released to public. *OCMG* [3] provides on-line Grid-enabled monitoring, compliant with Online Monitoring Interface Specification (OMIS). It requires manual code insertion to obtain arbitrary user-defined metrics.

3 eDragon Monitoring Framework

3.1 Goals

Rising complexity has made it harder to analyse the production environment and to work out the best deployment settings to use. Bearing this in mind, the eDragon Monitoring Framework (eDMF) was developed to monitor and measure the performance of Java middleware, track the underlying system and display all of this data in an easily digestible format for analysis. The eDMF can be used to spot any inefficiencies or issues of contention on the system as a whole and then later used to test different settings/changes which could solve the issue at hand. To gain full insight, the trace needs to be as complete as possible and should include any relevant details from the system resources (these have been identified as network, processing, and disk usage).

This approach is in stark contrast to other Java profilers/performance monitors since they generally focus on things from the application's perspective only. Measurements such as the memory usage, the thread processing times, thread synchronisation calls, object creation and the number of method invocations are the kinds of factors that are important to an application. While these other 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) [4] 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) [11] is a well known system developed to trace the processes running on a Linux OS, so its addition is needed to support non-Java applications. Paraver [7] 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. This has already been discussed in Section 2, where we compare other performance tools capabilities with our own.

3.2 JIS Details

To monitor any Java application using JIS it needs to be launched using a Bootstrap class. This Bootstrap class replaces the default ClassLoader with our own modified one and in this way allows us to modify the classes being loaded so that they can be traced. To prevent JIS from tracing ev-

erything blindly and thus adding unnecessary overheads, a configuration file is written to identify the classes and methods of interest. Javassist [5], a byte code modifying API, was used to help inject the instrumentation byte code easily.

When the natively coded jis library starts up initially, one of the first things it does is start the Linux Trace Toolkit (LTT). The LTT monitors system processes on a Linux operating system and requires patching the kernel so that there are hooks introduced for system operations. This has its own binary format and creates a file for each of the CPU's on the machine. In the future it will be necessary to update to LTTng (which has a different trace format) as the older LTT is no longer being developed. Viewing and analysing the final trace is left for Paraver.

3.3 Instrumentation Overhead

As with all instrumentation, the eDMF introduces overheads, but they have been kept to a minimum so as to not cause excessive interference with the normal running of the application. To try and get a figure on how much overhead there is with the eDMF, some experiments were done. In general the overhead introduced was around 15% of the total execution time using Java, but it depends on the number of events generated. Nevertheless, this overhead is low enough not to affect the conclusions extracted from standard applications analysis.

3.4 Future developments

Certain things that are being contemplated for future improvements of eDMF include providing different launch mechanisms, extending the JVMPI/JVMTI usage and porting to other operating systems besides Linux. The launch mechanism as it stands is based on a modified ClassLoader that needs to be put in place before the traced application is started. This is not always easy or practical so further investigation needs to be done to see what other options are worth considering. Having multiple ways to instrument the application would allow greater flexibility and the use of Aspect Object Programming (AOP) techniques could be a useful method for instrumenting the code throughout the application. The key points that would need to be instrumented would be written as a crosscutting concern for Aspect.

4 Problem analysis

4.1 Motivation

When we did some tests on a GT4 installation, trying the maximum number of jobs that we can 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 the system level or at the application (under a JVM) level.

4.2 GT4 Approach

The open source Globus Toolkit 4 (GT4) is a fundamental enabling technology for the Grid. The toolkit includes software services and libraries for resource monitoring, discovery and management, plus security and file management. Its core services, interfaces and protocols allow users to access remote resources as if they were located within their own machine space while simultaneously preserving local control over who can use resources and when.

We didn't use a complete installation of GT4, because we wanted to study the behaviour of Globus in a standard environment. To clarify, we basically used the service container (Java Core) [1] and the two WS-GRAM [2] services. This enabled us to do a remote execution of a simple job.

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 by using the axis classes.

WS-GRAM is basically two web services, the ManagedJobFactoryService, and the ManagedJobService (there are more services in WS-GRAM, but these two are the ones we will use). The first one enables the creation of a job resource when we want to submit a job to the remote machine, and the second one is the service that we have to call to start, stop, monitor,... the job. This is what we see from the client view. If we take a look inside the WS-GRAM, we can see how it manages to get the jobs executed on the remote machine using RunQueues threads. Each time we call the ManagedJobFactoryService, it creates a resource associated with our job and sends it to one of the RunQueues. Once the resource is in a RunQueue it goes through different states until it's sent to the local scheduler (for simplicity we use the fork scheduler provided with GT4 software). When the job is finished in the local scheduler, the job is sent to the last state. And with that, finally there is a response sent to the client.

4.3 Experimental environment

Our test environment is built up on two machines; a client and a server with similar features: 2-way, 2.4 Ghz Pentium Xeon, with 2 Gb of memory.

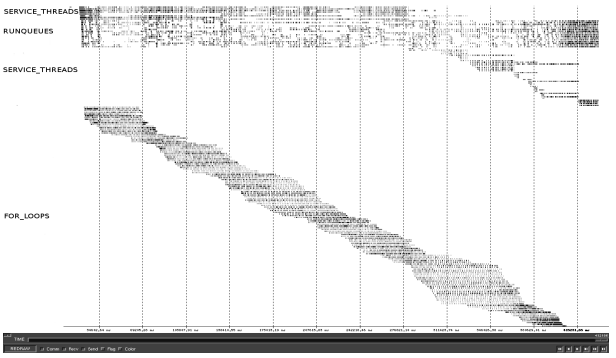

Figure 2. Paraver trace showing the system behaviour using overloading jobs and CPU assignment

4.4 Overloading the Server

In our test we have some jobs that can overload the CPU, by using a for-loop that creates 5 seconds of continuous 100% CPU load. In this case, we execute 128 of these jobs in parallel. This benchmark overloads the CPU at the system level, so it can produce the effect of reduced performance and adds inefficiencies at the middleware level, even losing jobs in some test cases (more than 150 submitted jobs). Figure 2 shows a Paraver window displaying the execution trace for this example. The horizontal axis represents the execution time in microseconds. The vertical axis shows the threads. As we can see from the trace, Globus continues trying to execute all the jobs as soon as possible (this is 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 times increase. The Operating System scheduler will spend more time on the jobs (because there is a larger number of them) than spending it on Globus and context switching contention is another problem also.

With Paraver's help, we can see that the job execution time lasts nearly 52 seconds on average (for a 5 seconds job). In the table 1 we show how this average decreases when we decrease the number of jobs executed. Note that we didn't include the 256 and 512 response times because the execution didn't fully complete in most cases.

From these traces, we can get some interesting views. Firstly, we can see how the O.S. distributes the CPU to the threads. Secondly, we can get the number of context switches of each thread (a total of 27925 and an average of 218 per thread) and the average burst time for example. We need some more basic results to make certain assumptions, so next we generate a 1 Job trace.

Taking a look at a 1-job trace using Paraver, we find a lower bound for the time of an execution, 5076 ms. Our proposal aims to get this response time (on the server side) for evey job when we submit 128 jobs. We also find that the amount of context switches during the execution of a single job stands at about 92.

# for-loop	Av. Response Time (s)	Jobs Executed
16	40.73/5.34	16/16
32	43.83/5.74	32/32
64	47.44/10.65	64/64
128	52.67/15.8	128/128
256	N/A	158/201
512	N/A	158/238

Table 1. Statistics from for-loop job execution with and without resource management (Original/Modified)

4.5 Results Analysis

In order to create a Resource Management policy, we compare the different parameters obtained. The most significative parameter could be the number of context switches. In order to reduce this, we should decrease the number of processes being executed on the system. This will likely produce a negative effect, because the number of threads relative to the system and middleware are decreased.

When the middleware layer (or O.S.) needs processing power it will typically take a loop job out of the processor; which will increase context switching. In order to reduce this, we can increase the priority of jobs to get a more continuous running time. To optimise resources, we should execute only as much jobs as are free processors on the server.

5 Resource management Analysis

Our simple proposal 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 switches over this process. If we reduce this number, we should improve the servers performance allowing it to spend more time doing useful things (also providing more CPU-affinity).

We are going to analyse the execution of 128 for-loop jobs with resource management. Now we are running only two jobs at once. Using the same method of statistical analysis as the previous traces, we can see how the average response time improves (Table 1), and how the context switching of the for-loop process decreases.

Another good improvement that can be found in Table 1, is a reduction of the average response time and increased the number of jobs executed. If we increase the response time in this policy, the number of jobs becomes greater and

therefore can be identified as a useful parameter for tuning an autonomic computing environment. Increasing the priority of the jobs (running 2 at once) can help us to achieve 200 finished jobs when we submit 1000 jobs, while the original middleware couldn't finish any. We greatly reduced context switches, one of the causes of system contention. We achieved a reduction on the 128 jobs test from 27925 in the original middleware to only 1788 with our resource management policy. On average every job does only 14 context switches (218 in the original middleware).

6 Conclusions

Grid Middleware is becoming popular and the Globus Toolkit provides us with the typical complexity of a middleware over a virtual machine. To get information about possible performance issues we should monitor all the layers of the system. Firstly, we should get knowledge about how the application level works (threads, levels and layers), continue with the JVM and get some internals (Garbage Collector, for example) and end at the system level to be able to see the interaction with operating system.

We have presented a framework that enables the monitoring and profiling of programs over different system/application layers, and we have also showed a survey of tools that enables profiling at different levels. 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. Since the objective of collecting execution information is to help produce system models, this framework design supports real-time correlation of the data collected from several levels which make up the application. This experimental application of the eDMF framework resulted in showing how it can help experts with the task of extracting knowledge to create management policies and how a management policy can be studied in detail, deducing further how it affects the performance of the system.

Finally, in the last section we showed how using a simple resource management policy can help to obtain an improvement on the server side. We improved several characteristics of the original globus middleware: we reduced average response time per job, reduced context switches over the system and most importantly, increased the number of jobs finished on the overloaded server. Although this resource management proposal is a simple one, our aim is to provide a way to analyse the Java middleware system using our monitoring framework and then improve it.

A monitoring framework can help us to see problems where there doesn't seem to be any. The problem that we identified in this paper could be difficult to find without the global monitoring framework. We needed information from the application level, the JVM level and the system level because Globus uses middleware that finishes running jobs at the system level. From that, we were able to discover some parameters that could possibly be used in an Autonomic Computing environment.

Without the eDMF this could be an impossible task. Improvements on this framework are part of our future work as well as continuing work on self-managing policies for the Grid Middleware.

Acknowledgement

This work is supported by the Ministry of Science and Technology of Spain and the European Union under contract TIN2004- 07739-C02-01. Thanks to Judit Gimenez for her help.

References

[1] G. Alliance. Globus toolkit documentation WS-Core. `http://www.globus.org/toolkit/docs/4.0/common/javawscore/`, 2005.

[2] G. Alliance. Globus toolkit documentation, WS-GRAM. `http://www.globus.org/toolkit/docs/4.0/execution/`, 2005.

[3] B. Balis, M. Bubak, W. Funika, T. Szepieniec, R. Wismüller, and M. Radecki. Monitoring grid applications with grid-enabled omis monitor. *Proc. First European Across Grids Conference, Santiago de Compostela, Spain*, pages 230–239, February 2003.

[4] 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.

[5] S. CHIBA. Javassist - a reflection-based programming wizard for java. *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[6] M. Harkema, D. Quartel, R. van der Mei, and B. Gijsen. Jpmt: a java performance monitoring tool. *In Proc. of TOOLS 2003, Urbana, Illinois, USA*, 2003.

[7] G. Jost, H. Jin, J. Labarta, J. Gimenez, and J. Caubet. Performance analysis of multilevel parallel applications on shared memory architectures. *International Parallel and Distributed Processing Symposium (IPDPS), Nice, France*, 2003.

[8] R. Nou, F. Julià, D. Carrera, K. Hogan, J. Caubet, J. Labarta, and J. Torres. Monitoring and analysis framework for java middlewares. *Research Report UPC-DAC-RR-2006-33*, 2006.

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

[10] B. Sotomayor and L. Childers. *Globus Toolkit 4 : Programming Java Services*. 2005.

[11] 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.