 DOI 10.1109/EMPDP.2005.47

WAS Control Center: An Autonomic Performance-Triggered Tracing Environment for WebSphere

David Carrera, David Garcia, Jordi Torres, Eduard Ayguadé, Jesús Labarta
{dcarrera, garcia, torres, eduard, jesus}@ac.upc.es

European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department, Technical University of Catalonia (UPC)
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034
Barcelona (Spain)

Abstract

Studying any aspect of an application server with high availability requirements can become a tedious task when a continuous monitoring of the server status is necessary. The creation of performance-driven autonomic systems can hurry up the analysis of this kind of complex systems. In this paper we present an autonomic performance-driven environment for WebSphere Application Server that can be used as the basis to construct systems that must monitor the performance of the system. As an applied use of this infrastructure, we present the WAS Control Center which is a deep tracing tool-set for 24x7 environments. It exploits the benefits of autonomic computing to lighten the costs of highly-detailed system tracing on a J2EE application server. The WAS Control Center is helping us in the creation of performance models of the WebSphere Application Server.

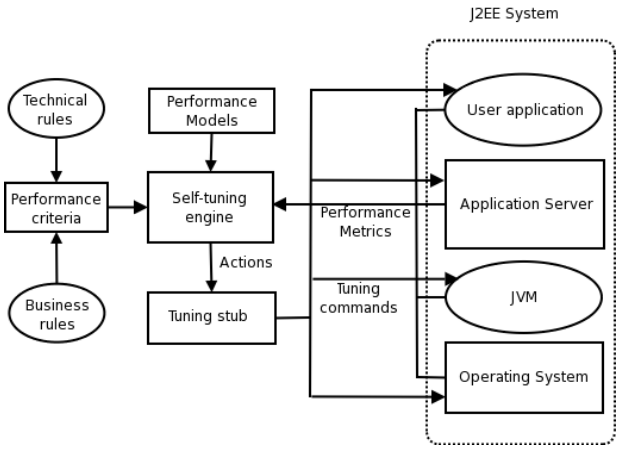


Figure 1. General overview of the WAS Control Center

1 Introduction

Application Servers based on the J2EE platform are widely extended. They are used in many commercial environments, in most of complex web applications currently online and in many B2B links. Facing up to the performance challenges that this platform involves is not a straightforward task in any way. The extremely complex execution environment of application servers provides a rich framework to develop and run web applications, but makes enormously difficult the task of studying and improving their performance. The parameters that affect the output level of an application server can change depending on many factors (i.e. the instantaneous workload on the server and the type of requests received can move the performance bottleneck of the server from the network bandwidth to the computing capacity). This dynamism increases exponentially the complexity of tuning these environments to obtain a maximum output level, particularly when it is done statically. It becomes specially true when the desired throughput of an application server is not defined in terms of technical units but in business profit units. And it is known that current trends of output level requirements are moving to the definition of Service Level Agreements[2].

An alternative to an inefficient static and manual tune of the application server is the creation of self-tuning application servers. It requires a solid knowledge of performance models for J2EE Application Servers and the techniques to make systems be in control of their own configuration parameters to achieve a maximum performance.

This paper describes our work done in the creation of an environment that is able to monitor the performance met-

rics of an application server, compare them to some user-defined rules and trigger the appropriate actions if necessary. We also describe how this environment has been used to create a performance-triggered autonomic tracing system (WAS Control Center) that can produce highly-detailed execution traces that are helping us in the definition of accurate performance models to describe the behavior of J2EE application servers.

The WAS Control Center environment is fully developed for the Java platform so it has no system dependencies. At the moment, the environment is dependent on the architecture of WebSphere Application Server[19] (WAS), so its use is limited to this application server. Future versions of this environment, based on the use of the Java Management eXtensions[20] API (JMX), will eliminate this dependency. The used middleware is WebSphere Application Server v4.0 Advanced Edition running on a 1.3 IBM Java Virtual Machine.

The environment described in this paper can be helpful on the construction of self-tuning systems. This is a first step toward the creation of a performance self-tuning engine for application servers, which is the final goal of our work. The work progress that must be done before achieving our objective consists of some complementary tasks. The study of the performance of application severs will drive us to the creation of performance models that, at the same time will be used to create the core of the self-tuning engine. The mechanisms to determine the stage of a performance model that defines the state of the application server must also be developed before an autonomic system can be in control of its own configuration. A general view of the components a system such as the described one should contain is shown in figure 1.

Similar proposals have been previously done on the area of distributed systems, like in [7] and in [10], but they are not focused on application servers like ours. The concept of autonomic computing has been widely explored and categorized by IBM, as described in [1]. There is some work published in the field of application servers' performance modeling, like in [11], in [12] and in [13], showing different approaches to the complexity of the problem. Also some works face up to the performance prediction for application servers, like in [14]. Some discussion about the creation of self-managed systems for web or application servers can be found, working with agents like in [9] or with other architectures, like in [16], in [15] and in [8].

2 Monitoring WebSphere with the Performance Monitoring Infrastructure

Some different approaches can be taken to continuously measure the performance of an application server. In the scope of this paper, we will focus on the measurement of the performance of the WebSphere Application Server, using the Performance Monitoring Infrastructure[5] (PMI) that it offers.

The Performance Monitoring Infrastructure consists in a set of libraries and packages developed to simplify the task of collecting, processing and visualizing performance information relating to the application server. PMI gets information from all the WAS components and makes it available to users. It offers a rich set of performance indexes of the application server. Some examples of indexes offered by PMI are the total number of requests to an object, response time of a web accessible object and number of concurrent active requests. These indexes can be obtained for both individual objects (servlets and EJBs) and the global system.

The WAS Control Center environment obtains continuous performance information from WebSphere by polling periodically the PMI Servlet deployed on it. This servlet, when accessed, queries the WAS Performance Monitoring Infrastructure to obtain performance indexes of the application server and returns them to the client summarized in a XML file that describes the current performance values for the different components of the application server. Each time the PMI servlet is accessed, an updated version of the XML-formatted performance report is returned. Other approaches we are working on are based on the use of the JMX API, what will make the environment independent of the application server.

The Performance Monitoring Infrastructure makes data available through an API composed of a set of interfaces. One remarkable interface of the PMI API in the scope of this paper is the PmiClient class, which can be used to develop PMI client applications, connected remotely to application servers. This interface creates an abstraction layer between client applications and the real Performance Monitoring Infrastructure.

3 Architecture of the WAS Control Center

The major feature of the WAS Control Center is the fact that is able to automatically detect performance problems on the application server according to a set of rules defined by the user, called hooks. Each hook refers to a performance index of the Application Server and sets a restriction to its value (maximum value, minimum value, deviation along time or other basic rules). If the hook restriction is violated, the WAS Control Center triggers an appropriate action. The performance indexes used to create hooks are extracted from the PMI of WebSphere. For instance, it is possible for a user to perform some actions when the number of active connections on the server exceeds 5000 clients or when the response time for a concrete Java Servlet or EJB is higher than a certain given value.

Although the WAS Control Center is composed of sev-

eral components, they can be mainly divided on two groups: the Control Center Application and the Remote Monitoring Server. The Control Center Application is in charge of monitoring the performance of WAS and of the applications deployed on it. This is done by accessing the WAS PMI servlet remotely and parsing the XML report of the current WAS performance metrics. The Control Center Application also implements a unified GUI for the overall environment. The Monitoring Server is a remote agent for the Control Center Application that is deployed on the JVM process running WebSphere. It can be used to take any required action on the application server machine in response to a hook violation command sent by the Control Center Application.

The general operation scheme of the developed environment begins with the startup of the WebSphere Application Server and the deployment of the Monitoring Server on WAS (in the same process). The Monitoring Server acts as a remote agent for the Control Center Application and must be initialized together with WAS. The Control Center Application will send commands that have to be executed on WAS to the Monitoring Server. The Control Center Application operation is directed by the WAS performance that is reported by the PMI. The PMI servlet is polled periodically by the Control Center Application and the obtained performance metrics are compared with the user-defined performance thresholds, named hooks. When the condition defined by any of the hooks is violated, the user-defined action associated with the hook is initiated. This is usually, but not only, translated to an action to be performed on the application server and the Monitoring Server is in charge of doing it.

In order to deploy the Monitoring Server on WAS, an automatic Java code interposition tool called JACIT (see section 4.1 for more details) is used to modify the startTransports() method of the WAS ServletEngine class and make it load the Monitoring Server before being executed. At the same time, the WAS Performance Monitoring Infrastructure is started up after the application server instance is completely loaded, and the PMI servlet (which makes accessible a XML-encoded representation of the PMI performance counters) is deployed on the application server and made accessible. With all this done, the Monitoring Server is ready to receive requests from the remote Control Center Application. These requests can be manually submitted using the environment GUI or can be an automatic response produced by the Control Center engine. A general operation diagram of an applied use of the WAS Control Center is shown in figure 2.

The Control Center Application works remotely from the application server machine. It is composed of four cooperative components: a GUI, an XML parser module, an application logic module and a communication stub.

The XML parser module is in charge of making this initial parsing process. The amount of information from the XML file obtained from the PMI servlet that is offered to the user depends on the configuration of the WAS Control Center. So, the XML parser module not just parses the XML file but also selects what information from the XML file is finally shown to the user and used to define hook restrictions. As long as the tree structure of the XML file generated by the PMI servlet is dynamic (each time the PMI servlet is accessed the produced XML file can change depending on the activity of the server), the XML parser has to detect the changes on its structure and adapt it to the in-memory representation.

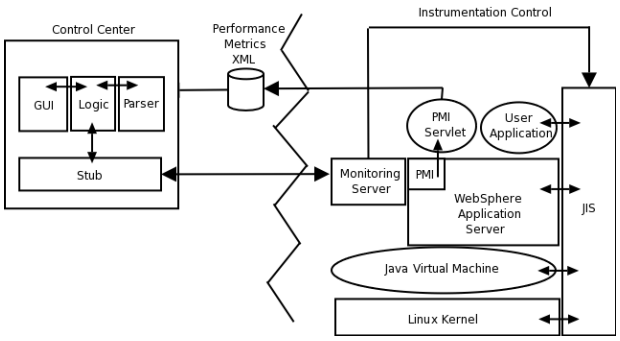


Figure 2. WAS Control center operation diagram for autonomic system tracing

The logic module of the Control Center Application contains all the program logic required to make the presented environment work. This logic module is configured to periodically request the PMI servlet to generate an XML report of the las observed performance metrics on the application server. The request rate to the PMI servlet is a user-configured parameter. Each time the XML-formatted file is obtained through the PMI servlet, all the hooks restrictions are checked.

The GUI module offers an overall configuration interface for the entire environment. On figure 3, an example of the Control Center GUI can be observed. On it, the last observed value for the number of concurrent requests for the servlets of the DefaultApplication is shown. For this metric, the mean value and the integral value can be used by the user to define hook restrictions.

4 Using the WAS Control Center for autonomic system tracing

Application servers (specially those of the complexity and robustness of WAS) are usually executed in environments with high availability requirements. In many cases this means ”24x7” clusters of fault-tolerant systems with high-availability requirements. In this kind of environ-

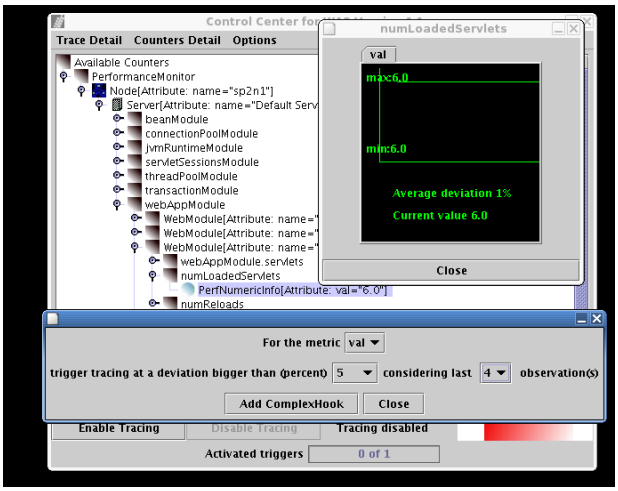


Figure 3. View of the WAS Control Center GUI

ments, obtaining detailed execution information to construct system performance models can be a problem because tracing tools are not overhead-free and they should not be working continuously.

The presented development tries to overcome these limitations. We use the WAS Control Center environment to control a highly-detailed system tracing tool in an autonomic way. It allows the creation of system execution traces only during the performance degradation to allow them to make a subsequent post-mortem analysis.

4.1 Tracing and analysis tools

Three previously developed tracing and analysis tools are integrated with the WAS Control Center to create an autonomic tracing environment. These characteristics of these tools are briefly discussed below.

JIS: Highly-detailed system tracing

The system tracing of an Application Server can be done using JIS[3] (Java instrumentation Suite), a tracing environment oriented to the study of Java applications and specially J2EE Application Servers. Four levels are considered by JIS when tracing a system: operating system, JVM, middleware (application server) and application. Information collected by all levels is finally correlated and merged to produce an execution trace file. The level of detail of the information produced by each JIS level can be dynamically configured.

Paraver: Trace analysis and visualization

Paraver[21] is a flexible performance visualization and analysis tool based on an easy-to-use Motif GUI. Paraver was developed to respond to the need to have a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information useful to improve the decisions on whether and where to invert the programming effort to optimize an application.

JACIT: Code Interposition based on Aspect Programming

The JACIT tool (Java Automatic Code Interposition Tool) can be used to apply the aspect[17, 18] programming paradigm to the modification of existing bytecodes of an application without need of source code availability.

With the JACIT tool it is possible to open a jar file from any application, choose one of the classes contained in the jar file, select one of the methods or interfaces of the method and decide to add some code before or after invoking it. The inserted code can use anyone of the parameters of the Java method. Later, the code can be compiled to test its correctness and after that, an equivalent aspect programming file is generated (if wanted) and the needed changes are applied to the jar file to execute the added code when required. Finally the jar file is saved and a backup jar file is also produced.

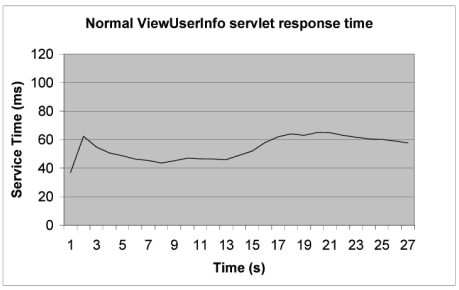
4.2 Triggering the system tracing

When the value of one of the performance metrics reported by the PMI servlet violates one of the defined hook restrictions, the JIS system tracing process is enabled and kept in this state until the reported values fall again inside limits defined by the broken hook.

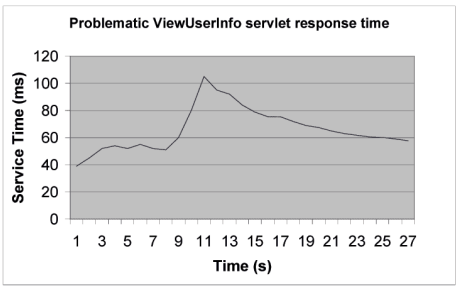
5 Testing the autonomic system tracing environment

In this section we present an example of use of the WAS Control Center applied to the system tracing environment. The chosen testing system is composed of three machines: a 4-way application server machine running WebSphere 4.0, a 2-way database host machine running MySQL and a client machine running the servlet-based version of the RUBiS[4] 1.4 benchmark. Each one of these machines disposes of 2Gb of physical RAM. The MySQL database server is hosting the tables required by the RUBiS benchmark servlets running on WAS.

The RUBiS benchmark simulates an Internet auction site and is distributed in some different versions, depending on its implementation: PHP, Servlets or EJBs. We chose the Servlets version for our experiment. The benchmark uses a database to store information relative to the products, users and bids.



(a) Normal response time during a benchmark run



(b) Degraded response time

Figure 4. Observed response time for a servlet of the RUBiS Benchmark

As an experiment, we put this environment to work and waited it to detect some performance problems without human cooperation. After a system warm-up, we measured the response time for one RUBiS servlet (ViewUserInfo). The observed response time along time can be seen on figure 4(a). At the moment, all the system metrics seemed stable, so we decided to set the performance metrics thresholds (hooks) on the WAS Control Center, setting it up to trigger JIS when the observed response time for the ViewUserInfo servlet varied $\pm 10\%$ of the normal value. Although it was done for the response time of the ViewUserInfo servlet, it could have also been done for the entire application server or for some other metrics.

Short time after having launched the benchmark, the response time shown by the PMI servlet for ViewUserInfo servlet increased. The observed response time when the performance degradation was detected can be seen on figure 4(b). It can be observed on this figure that the response time for the servlet called ViewUserInfo shows an important peak value in comparison to the normal execution behavior. This fact caused the WAS Control Center logic module to trigger the JIS system tracing process on the application server machine. With the system tracing that JIS performed on the WAS system, we expected to detect the cause of the observed performance problems.

The analysis of a trace file generated by JIS starts by opening it whit Paraver. This first visualization of the trace file didn't provide too much information about any performance degradation on the service of servlets on the application server. To get this information we had to consider using the statistical tools of Paraver. The first thing to do was to study the time distribution of the operations performed by the servlet which caused the triggering of the tracing process. This servlet, as commented above, is the ViewUserInfo RUBiS servlet.

The first analysis on the time distribution of the servlet execution just considered the activity on the system caused by the servlet which triggered the instrumentation process. We can configure the Paraver visualization module to only show information concerning to the ViewUserInfo servlet. So, we could study what were threads doing during the service of each request received for the ViewUserInfo servlet. Concretely, we were interested on getting information about the state of each thread of the WAS Servlet Engine thread pool that had been in charge of the service of at least one of the requests to the ViewUserInfo servlet.

By studying the results, we could see that most of the execution time for each request of the servlet during the system low performance period had been spent in the interruptible blocked thread state, independently of which thread was servicing the request. This situation was suspicious of either a problem of contention on the application server resources (i.e. locks or database connection pools) or a problematic blocking system call.

The next logical step was to start investigating possible problems with system calls. The result of this analysis indicated that the highest percentages of service time for the servlet processing were spent on some socket receive system calls. After a new study on the socket channels affected by these blocking operations (which is beyond the scope of this paper), it was determined that they corresponded to the database connection pool channels. After obtaining these results, we decided to study the activity of the database server machine during the benchmark run, and it was determined that the intense use of the machine by one of its users caused the increase on the response time of the database server. Anyway, the tracefile contains information enough to work intensively in the creation of new perfor-

mance models describing the relation between the performance of the database server and the application server.

Although this is a simple testing example for the environment and that our purposes were centered on obtaining valuable information about the factors that can modify the performance of an application server, the experience shown us that it could be a very useful environment for system administrators which are in charge of ”24x7” environments and that need highly detailed information about their systems when they seem to be in trouble.

6 Conclusions & Future Work

In this paper we have shown a performance-driven environment for WebSphere that can be used as the basis for the construction of autonomic middleware systems based n WAS. As a first use of this environment, we have applied it to our research necessities and it has been integrated with some previously developed tools to control a fine-grain system instrumentation process. The results of this association are helping us in the construction of performance models for J2EE application servers.

Our development represents a first step toward the creation of a self-optimizing and self-tuning environment for WebSphere. We have started with the creation of a performance-sensitive environment an its application to an autonomic tracing system. This environment will be our start point to create more advanced autonomic systems, up to create a generic self-tuning engine for J2EE application servers.

The combination of an instrumentation tool, a visualization tool and a remote control tool is now probed to be enough to have a ”24x7” deeply traced server without need of human cooperation until the analysis step.

6.1 Future Work

The framework described on this paper, at the moment,is WAS specific as far as it depends on the use of the PMI to collect performance metrics. Our work now is centered in moving to a new technology: the JMX core of WAS 5 and JBoss 3/4. This Java API, which stands for Java Management eXtensions, can be used to create manageable components and integrate them as an application. With this technology, each configurable component of an application server is represented by a MBean object (Management Bean) and can be remotely managed. To support different remote access techniques, a JMX adaptor layer is incorporated, which usually supports HTTP/SOAP, RMI and Bean accesses.

Acknowledgements

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) and CEPBA-IBM Research Institute (CIRI) for supplying the computing resources for our experiments. This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01.

References

[1] J. O. Kephart and D. M. Chess. The vision of autonomic computing. IEEE Computer, 36(1):41–50. IEEE, January 2003.

[2] Keller, A., Ludwig, H., The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services, Journal of Network and Systems Management, Special Issue on ”E-Business Management”, Volume 11, Number 1, Plenum Publishing Corporation, March, 2003

[3] D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta: An Instrumentation Environment for Java Application Servers, 2003 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 166-176, March 2003.

[4] C. Amza, A. Chanda, E. Cecchet, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel: Specification and Implementation of Dynamic Web Site Benchmarks, Fifth Annual IEEE International Workshop on Workload Characterization (WWC-5), November 2002.

[5] S. Rangaswamy, R. Willenborg, and W. Qiao: Writing a Performance Monitoring Tool Using WebSphere Application Server’s Performance Monitoring Infrastructure API, IBM WebSphere Performance, IBM WebSphere Developer Technical Journal, February 2002

[6] D. Viswanathan and S. Liang: Java Virtual Machine Profiler Interface. IBM Systems Journal, 39(1):82-95, February 2000.

[7] K. P. Birman, R. van Renesse, J. Kaufman and W. Vogels: Navigating in the Storm: Using Astrolabe for Distributed Self-Configuration, Monitoring and Adaptation, Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services (AMS’03) June 25 - 25, 2003

[8] Hoi Chan et al., Approach to Policy Execution In Autonomic Manager Toolkit, First Workshop on Algorithms and Architectures for Self-Managing System, Federated Computing Research Conference 2003

[9] Y. Diao, J.L. Hellerstein, S. Parekh and J.P. Bigus, Managing Web Server Performance with AutoTune Agents, IBM Systems Journal, Vol 42, No. 1, 2003.

[10] D.J. Kerbyson, J.S. Harper, E. Papaefstathiou and G.R. Nudd, Use of Performance Technology for the Management of Distributed Systems, 6th International Euro-Par Conference (Euro-Par 2000), Lecture Notes in Computer Science 1900, Springer Verlag, August 2000, pp.149-159.

[11] J.C. Hardwick, E. Papaefstathiou, and D. Guimbellot, Modeling the Performance of E-Commerce Sites, Journal of Computer Resource Management, Winter 2002 Edition, Issue 105, pp. 3-12.

[12] D.Bacigalupo, J.D.Turner, S.A.Jarvis, G.R.Nudd. Modelling Dynamic e-Business Applications Using Historical Performance Data. 19th Annual UK Performance Engineering Workshop (UKPEW03), University of Warwick, UK, 9-10 July 2003, pp 352-362

[13] S. Kounev and A. Buchmann. Performance Modelling of Distributed E-Business Applications using Queueing Petri Nets. 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS03), 2003.

[14] Yan Liu, Ian Gorton, Anna Liu, Ning Jiang, Shiping Chen, Design a Test Suite for Empirically-based Middleware Performance Prediction, TOOLS PACIFIC, Feburay 2002

[15] M. Trofin and J. Murphy. A Self-Optimizing Container Design for Enterprise Java Beans Applications. 8th International Workshop on Component-Oriented Programming WCOP2003, (in conjunction with ECOOP'2003), Darmstadt, Germany.

[16] A. Mos and J. Murphy. Performance Management in Component-Oriented Systems using a Model Driven Architecture Approachm, 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC, September 2002, Lausanne, Switzerland).

[17] John Viega and Jeffrey Voas: Can Aspect-Oriented Programming Lead to More Reliable Software?, in the November/December 2000 issue of IEEE Software

[18] Markus Voelter: Aspectj-Oriented Programming in Java, in the January 2000 issue of the Java Report.

[19] IBM WebSphere Application Server
http://www.ibm.com/websphere

[20] Java Management eXtensions
http://java.sun.com/products/JavaManagement/

[21] Paraver
http://www.cepba.upc.es/paraver