PERFORMANCE IMPACT OF THE GRID MIDDLEWARE*

DAVID CARRERA[1] , JORDI GUITART[2] , VICENÇ BELTRAN[3] , JORDI TORRES[4] , AND EDUARD AYGUADÉ[5]

[1] Technical University of Catalonia (UPC), Barcelona (Spain), dcarrera@ac.upc.edu,
[2] Technical University of Catalonia (UPC), Barcelona (Spain), jguitart@ac.upc.edu,
[3] Technical University of Catalonia (UPC), Barcelona (Spain), vbeltran@ac.upc.edu,
[4] Technical University of Catalonia (UPC), Barcelona (Spain), torres@ac.upc.edu,
[5] Technical University of Catalonia (UPC), Barcelona (Spain), eduard@ac.upc.edu

Abstract. The Open Grid Services Architecture (OGSA) defines a new vision of the Grid based on the use of Web Services (Grid Services). The standard interfaces, behaviors and schemes that are consistent with the OGSA specification are defined by the Open Grid Service Infrastructure (OGSI). Grid Services, as an extension of the Web Services, run on top of rich execution frameworks that make them accessible and interoperable with other applications. Two examples of these frameworks are Sun's J2EE platform and Microsoft's .NET.

The Globus Project implements the OGSI Specification for the J2EE framework in the Globus Toolkit. As any J2EE application, the performance of the Globus Toolkit is constrained by the performance obtained by the J2EE execution stack This performance can be influenced by many points of the execution stack: operating system, JVM, middleware or the same grid service, without forgetting the processing overheads related to the parsing of the communication protocols. In the scope of this chapter, all this levels together will be referred to as the grid middleware.

In order to avoid the grid middleware to become a performance bottleneck for a distributed grid-enabled application, grid nodes have to be tuned for an efficient execution of I/O intensive applications because they can receive a high volume of requests every second and have to deal with a big amount of invocations, message parsing operations and a continuous task of marshaling and unmarshalling service parameters. All the parameters of the system affecting these operations have to be tuned according with the expected system load intensity. A Grid node is connected to to other nodes through a network connection which is also a decisive factor to obtain a high performance for a grid application. If the inter-node data transmission time overlaps completely the processing time for a computational task, the benefits of the grid architecture will be lost. Additionally, in many situations the content exchanged between grid nodes can be considered confidential and should be protected from curious sights. But the cost of data encryption/decryption can be an important performance weak that must be taken into account.

In this chapter we will study the process of receiving and executing a Grid job from the perspective of the underlying levels existing below the Grid application. We will analyze the different performance parameters that can influence in the performance of the Grid middleware and will show the general schema of tasks involved in the service of an execution request.

Key words. Grid middleware, Web Services, Globus, J2EE, Performance analysis

AMS subject classifications. 68M14, 68W30, 68W10, 68W15

1. Introduction. The Grid infrastructure provides some integrated technologies thought to make possible the creation of distributed applications that can overcome the currently existing limitations related to issues such as quality of service, security, resource discovery or uniform access. But these technologies require the creation of open and standard interoperation mechanisms to support the wide range of applications and systems that the grid is made up of.

The creation of open standards for heterogeneous systems interoperation requires the existence of a set of application-level protocols that make possible the exchange of information across systems and the existence of portable execution environments sup-

*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 and by the CEPBA-IBM Research agreement.

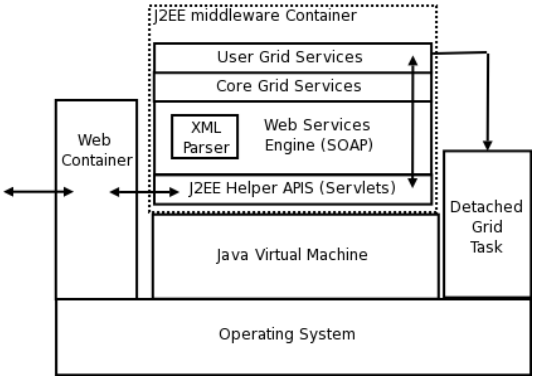

FIGURE 1. Execution environment for a grid middleware based on J2EE

porting these protocols. The joint of theses protocols and the corresponding execution environments is known as a Grid middleware.

There are several efforts to consolidate a general architecture for the Grid, considering the exposed services and resources. Some of these efforts have failed but other ones have grown to currently become successful projects. There are open source initiatives, government funded projects and proprietary solutions. Some examples of Grid middleware ongoing projects can be found in [7].

One of these initiatives is the Globus Alliance[3], which is a joint project between some research centers, universities and other organizations to create fundamental technologies supporting the Grid. Their major contribution is the specification of the Open Grid Services Architecture[20] (OGSA) and the Open Grid Service Infrastructure [5] (OGSI). These specifications define a global architecture for the Grid and a set of interfaces consistent with the architecture. The first implementation of the OGSI specification is the Globus Toolkit[44], which is composed of several services and libraries offering a development and execution framework to create and deploy Grid applications.

As it was mentioned before, the Grid is composed of a number of heterogeneous systems and all them have to communicate following common protocols. The Grid middleware supports these protocols and provides an execution environment for Grid applications, but it must be a portable piece of software if it has to be executed in very diverse systems. For this reason, the Globus Alliance has chosen widely adopted portable technologies to create and support their Grid middleware and protocols. The Globus Toolkit (on its version 3 and following) is based in the Java[41] technology as the underlying execution environment for the Grid middleware software and relies on the Web Services[47] technologies to communicate the systems that integrate the Grid. The Java platform is a portable programming language and virtual execution environment supported by most of the existing systems. The Web Services technologies are continuously under development and define a set of basic standards required to perform global interoperation between heterogeneous systems. A brief diagram for the working components of the Globus Toolkit 3 (GT3) can be seen in figure 1.

The Web Services technologies are supported by several applications and libraries. They rely on the common internetworking infrastructures most widely extended nowadays, such as TCP/IP and HTTP[32]. A usual deployment scenario (but not the only one) for applications and middleware technologies that deal with Web Services technologies are J2EE[40] application servers. These kind of environments are used as an execution framework that isolates applications from dealing with networking issues (receiving connection requests, managing sessions, exchanging HTTP messages...). In the Grid, they are used as the supporting platform to create Grid middlewares. The Grid middleware can be seen as a web (or J2EE) application running in the application server. When this is the scenario, the concept "Grid middleware" can be extended to fit the entire execution stack: from the operating system to the Web Services applications, covering the Java Virtual Machine and the Application Server layers.

When a Grid node receives several requests (or jobs in Grid terminology), the overhead introduced by the Grid middleware (as defined above) can have a noticeable impact on the overall execution performance. Managing connections, parsing requests and dealing with the Web Services protocols are tasks that require a non negligible amount of computing resources. Tuning the environment, and specially the application server used as the Grid middleware framework, is not a trivial task but it is a crucial step in order to improve the quality of service offered by a grid node.

In this chapter we will describe the execution environment of the Globus Toolkit middleware, running in the framework offered by a widely extended open source J2EE application server (Tomcat[29]). We will study the process of receiving and executing a Grid job from the perspective of the underlying levels existing below the Grid application. We will analyze the different performance parameters that can influence in the performance of the Grid middleware and will show the general schema of tasks involved in the service of an execution request. The following of the chapter is organized as follows: section 2 describes the technologies and concepts associated with the grid middleware architectures considered in this chapter, section 3 discusses which is the best approach for the performance analysis of a J2EE grid middleware as well as references some tools that can help users in this task, section 4 discusses in detail some of the factors that produce a higher influence in the performance of a grid middleware, section 5 covers some of the currently active research trends in the area of performance improvements for J2EE middlewares and, finally, section 6 summarizes all the information presented in this chapter and introduces some final remarks about the impact caused by the grid middleware in the global grid performance.

2. Grid middleware fundamentals. In this section we will discuss which are the basic technologies involved in the creation of a grid middleware based in the Web Services technologies, and will detail the execution steps required to invoke a grid service deployed in a J2EE container.

2.1. Grid services and web services. A Web Service[47] is an interface that describes a collection of operations (and the implementation of these operations) that are accessible through a network. Each Web Service is described using an standard notation known as service description, which gives details about the format of the messages accepted by the web service, the transport protocols supported and the location of the service. The service description hides the complexity of the implementation of the Web Service, so it can be implemented in any existing platform that is able to process the messages detailed in the service description and makes possible the decoupling between the service description and the service implementation.

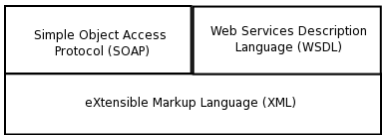

FIGURE 2. Web Services basic protocols

The technologies used to access and invoke Web Services are well known and widely used and take the XML language as their basis, as it can be seen in figure 2. The eXtensible Markup Language[2] (XML) is a markup language, derived from SGML, used to represent data using textual information. The use of XML is widely extended, and it is the technology on the basis of which two other fundamental elements for the Web Services are constructed: the Web Services Description Language[16] (WSDL) and the Simple Object Access Protocol[12] (SOAP). Each Web Service is described by a WSDL document, where the Web Service developers determine the operations that are invocable in a Web Service as well as the format of the invocation, parameters and results that are supported by the Web Service. The SOAP protocol is basically a remote invocation technology based in XML. This protocol is used to invoke remote procedures using XML as the encoding language for the invocation descriptor and to marshall the parameters and results. SOAP messages are usually sent across the Internet using the HTTP transport protocol.

A Grid Service[5][20], as understood in the context of the Globus Toolkit, is a service compliant with the Open Grid Services Infrastructure specification (OGSI) and that exposes itself through a Web Services Description Language[16] (WSDL) document. In consequence, it is a general Web Service adapted to the specific requirements of Grid Applications.

The OGSI specification uses the Web Services technologies to expose primitives that make possible the creation and composition of Grid Services. So the interfaces defined by the OGSI specification to be exposed are not services themselves but the mechanisms that can be used to construct services. On this way it is guarantied that all Grid Services will have a common core of functionalities independently of their semantics, such as discovery, dynamic service creation, lifetime management, notification and manageability.

2.2. Executing web services in a J2EE environment. The environment where a Web Service can be deployed and invoked is known as a Web Services engine. It is in charge of dealing with the protocols related to the Web Services technology (mainly SOAP and WSDL). The SOAP protocol is usually bound to the HTTP protocol and it leads to the requirement of implementing a Web server container (or any HTTP-enabled component at least) inside of the engine, or to the need of integrating the Web Services engine into an existing HTTP supporting environment. This condition converts any container framework (such as J2EE[40] or .NET[31]) into an execution environment specially appropriate to support a Web Services engine.

The Globus Toolkit 3 can be deployed on top of Tomcat[29] and uses it as the hosting environment to run the Web services engine. Although GT3 incorporates an embedded web container, it is only recommended, according to the documentation, for testing purposes. The hosting environment for production deployments is a complete web container such as Tomcat. When GT3 is deployed as a dynamic web application in the Tomcat container, it takes advantage of the optimized Tomcat architecture to deal with the Web transport protocols. GT3 uses Apache Axis[43] as its Web

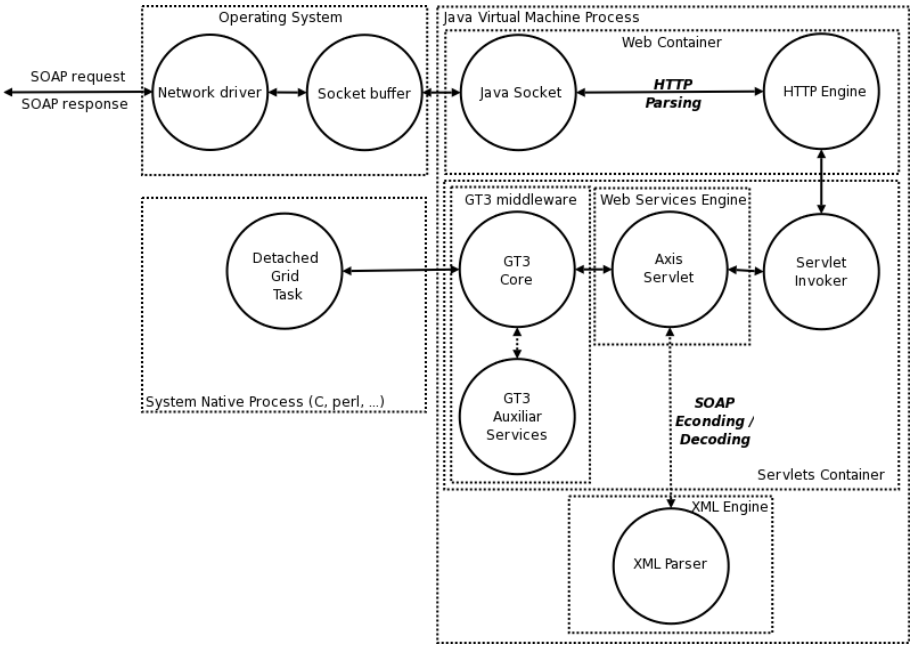

FIGURE 3. Simple GT3 invocation diagram

Services engine, which is an implementation of the SOAP protocol that supports Web Services technologies. It is supplied as a web application based in the Java Servlets technology. The Java Servlets[42] specification (which is a part of the J2EE platform) defines a server-side scripting technology created to extend the dynamic properties of Web servers. It can be thought of as a light piece of software that can be invoked remotely using the Web server infrastructure and that dynamically produces a response that is sent back to the invoker. Tomcat is the servlet container used as the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. It is developed and maintained by the Jakarta project[4], which is a part of the Apache Software Foundation and is a full Java application.

The overall simplified operation diagram and the software components involved in the invocation of a remote Grid Service using GT3 can be seen in figure 3. The invocation begins when a SOAP request (properly encapsulated in a HTTP message) is received in the grid node. The request is moved from the network to the corresponding system socket and then the corresponding JVM socket will make it available to the Java application. At this point, the execution of the grid middleware stack begins. First, the web container processes the HTTP request and decides, according to the context indicated by the requested URL, that the information contained in the HTTP message must be processed by the Web Services engine (the Apache Axis servlet in the scope of this chapter). The servlet manager of the Web container (Servlet invoker in the diagram) redirects the request input stream to the Axis servlet to be processed as a SOAP request. The Web Services engine decodes the incoming SOAP request (with the help of a complementary XML parsing engine) and determines which Web Service must be invoked. The Web Service is then invoked (in the case of the GT3, a Core servlet is invoked) and it uses its auxiliary services helpers to process the request, ensure the necessary encryption and to instantiate the corresponding native process

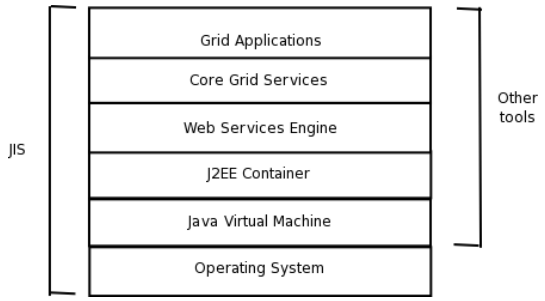

FIGURE 4. Execution stack levels considered by JIS

(a perl process, native compiled or even other Java services).

Finally, it is remarkable that the OGSI specification is not only being implemented for the J2EE platform, and it is also being covered by other projects that are implementing it using other container frameworks, like the OGSI.NET[46] project is doing.

3. Performance analysis tools. The performance offered by a J2EE middleware is constrained by the performance offered by all the levels involved in its execution stack. From the application code to the low level system kernel, all the layers of the stack influence the overall performance of the system. This means that studying the performance of a J2EE environment for tuning purposes requires the use of tools that can cover the entire execution stack and moreover, to correlate the information obtained from each one of the layers to create a global performance vision.

Different approaches are used on existing tools to carry on the analysis process of Java applications. All of them report information about the behavior of the applications in terms of object creation, loop structures, execution patterns or time consumption inside functions. However, they do not consider system status nor the interaction between the applications and the underlying operating system. Some tools oriented to study application servers report different metrics that measure the application server performance, collecting information through the JVM Profiling Interface (JVMPI [45]). This limits the kind of information that they can get and therefore, their ability to perform a fine-grain analysis of the multithreaded execution and the scheduling issues involved in the execution of the threads that come from the Java application. The basic actions performed by current web-based technologies are the following: reading/writing contents from/to disks; receiving/sending data from/to networks; and finally, processing data coming from disks and networks. This implies that, mainly, three basic system resources support web-based technologies and applications: processors, disks and networks. Thus, a general requirement in order to perform complete performance analysis of web-based technologies and applications is to be able to obtain detailed information about the usage of these resources. Some examples of these tools are JProbe[34] from Quest Software, Jprofiler[18] from ej-technologies, OptimizeIt[11] from Borland Corporation or WebSphere Studio Profiling Tool[27] from IBM.

A different approach is followed by the Java Instrumentation Suite (JIS [15], from the eDragon research group, see [1]), which collects information from all the levels involved in the execution stack of J2EE applications, and correlates the data obtained from each source (application, JVM and system). The produced information

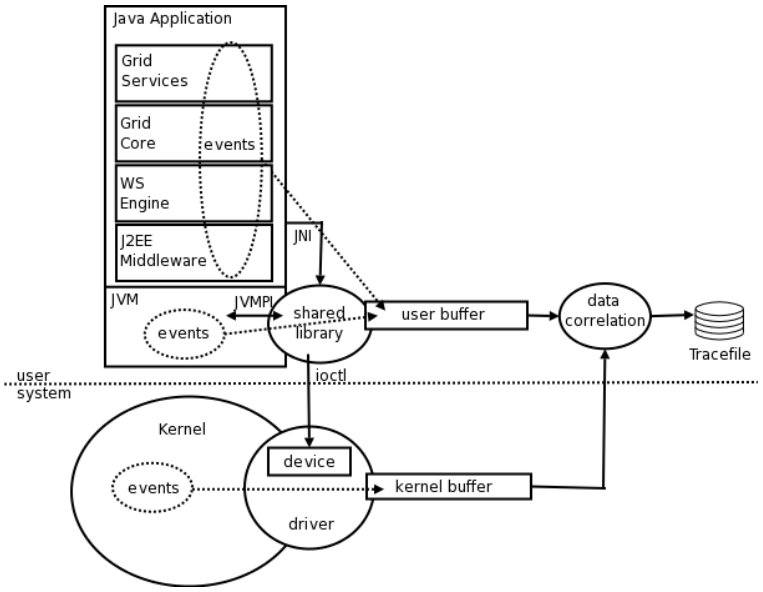

FIGURE 5. JIS operation procedure

is dumped to a tracefile and can be studied later with the appropriate trace analysis tools (Paraver [19], for instance). Lastly, a new instrumentation tool that fits the structure of JIS has appeared in the market, which is named PerformaSure[35], from Quest Software.

JIS focuses on Thread status, I/O operations (over storage devices as well as sockets), network devices and memory usage. The interaction between application and system can be studied to see how the use that the application makes of the system (i.e. number of threads created, degree of activity of these threads, amount of I/O operations done or usage of network resources) affect the resulting performance of applications. All the information sources mentioned above can be classified in different levels depending on how they can be accessed to obtain data. JIS instruments a Java application at three different levels: Application Level, Java Virtual Machine level and System level, as shown in figure 4. This multi-level architecture is justified by the different nature of data sources implied in the instrumentation process.

Information collected by the different layers conforming JIS is finally merged in order to produce a unified trace. The following subsections describe in detail the architecture of the different JIS levels for the IA32 Linux implementation, which is represented in figure 5 and detailed in following subsections. For a more in depth description of JIS, the reader can refer to [15].

3.1. Application level. This is the highest level considered by JIS and it considers all the J2EE application. In the case of a grid middleware, which is deployed on top of the J2EE middleware, it covers all its layers as it is shown in figure 5. The objective of this level is to introduce some especial events into the final tracefile in order to provide it with the appropriate grid middleware semantics. These events can be produced directly from the application code (needing its recompilation) or using helper applications such as JACIT[14], which allows the insertion of code in a Java class file, without need of recompilation, using the Aspect programming paradigm.

The application level instrumentation is specially useful when working with a J2EE application like GT3 (or any similar grid middleware based in the Java platform) because events can be inserted, for instance, before and after processing a job or when a new request is received by the Web services engine in order to make this information available in the output tracefile, correlated with the information produced by the other JIS levels and which can result very helpful for the following analysis steps.

3.2. JVM level. Java semantics are only understood from inside the JVM. Because of this, comprehensive instrumentation of Java applications must be composed, in part, by internal JVM information. Current versions of JVM implement a Profiler Interface called JVMPI[45] that is a common interface designed to introduce hooks inside JVM code in order to be notified about some defined Java events. This facility is used by JIS to include information about Java application semantics on its instrumentation process. This means that a developer analyzing own applications will be able to see system state information during execution expressed in relation with some of the developed Java application semantics. The JVMPI is based on the idea of creating a shared library which is loaded on memory together with the JVM and which is notified about selected internal JVM events. Choosing hooked events is done at JVM load time using a standard implemented method on the library that is invoked by the JVM. Events are notified through a call to a library function that can determine, by parsing received parameters, what JVM event is taking place. The treatment applied to each notified event is decided by the profiler library, but should not introduce too much overhead in order to avoid slowing down instrumented applications in excess. Some of available events are: start and end of garbage collecting, class load and unload, method entry and exit and thread start and end.

3.3. System level. To perform a useful application instrumentation, continuous system state information must be offered to developers. On the Linux version of JIS, and taking into consideration the open source characteristics of Linux systems, system information is directly extracted from inside kernel. This task is divided in two layers: one based on a kernel source code patch and the other on a system device and its corresponding driver (implemented in a Linux kernel module). The kernel module provides some basic JIS functionalities, such as the interception of system calls, implementation of a device driver for instrumentation control or the creation of an user space system instrumentation control interface through the ioctl system call. Each one of this system activities are reflected in the final tracefile in the form of an event, that later can be studied in relation to the events produced by all the other JIS levels. There is also an alternative version of JIS which uses LTT[33] as the system level tool that provides information about the system status continuously, instead of relying in the own kernel patch and module.

4. Grid middleware performance. In this section we will study the performance issues affecting the global behavior of the grid execution stack of a Grid middleware based on Web Services such as the Globus Toolkit 3. The improvable points with a major impact in the global server performance can be seen in figure 6.

4.1. Web services engine: SOAP and XML. The use of SOAP in high performance scientific computing environments provides a new level of interoperability in the area but at the price of introducing a new degree of inefficiency in communications. Distributed scientific computing applications usually exchange large arrays of floating point numbers. The representation of information in the XML language can need as much as 10 times the amount of room that the same information requires to

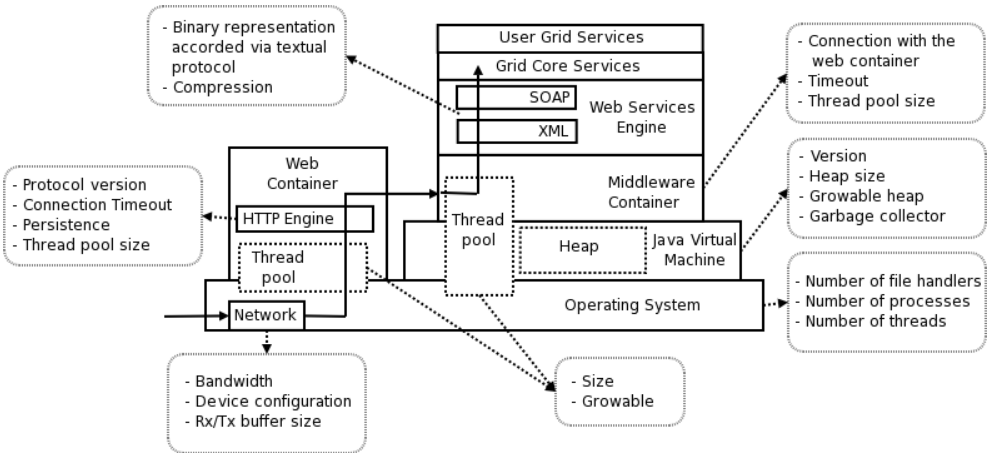

FIGURE 6. Tuning points in a grid middleware

be represented in the corresponding binary format, so, in order to increase the performance of the transport protocols in charge of moving XML documents between hosts, a point of study is the impact of representing XML data using non-text formats. This topic has been studied in [36], [37] and [13] as a generic technique, and in [22] when it is applied to the Web Services technologies. These studies conclude that the use of text compression techniques for the XML language can lead to a dramatic increase of the performance of the XML based applications.

An alternative to text compression is discussed in [17], where the authors state that in order to achieve a higher level of performance from the SOAP protocol, it must be extended to support binary representations of scientific numbers or it must be used only as an interoperable mechanism to establish a negotiated binary communication protocol. Similar proposals are done in [38] and [39], and the W3C has already created a working group to study under which situations, the use of binary representations of a XML document can be beneficial, as it is discussed in [6].

4.2. Web and middleware Container. Dynamic Web applications, such as the case of a Grid middleware like GT3, are client/server applications that are run on top of a middleware framework and that require a Web container to be in charge of dealing with the Web transport protocol (HTTP) and the underlying levels (TCP and IP), and a middleware container (such as J2EE or .NET container) used to create a rich execution framework for the application components deployed on it. In the case of GT3, the Grid middleware core is deployed as a set of Java Servlets and the additional resources required.

The Web container and the J2EE container can be hosted in different machines or in the same. Even the J2EE container can be distributed across multiple servers. Anyway, they work cooperatively to service client requests dynamically. The Web container accepts incoming connection requests, parses the HTTP protocol to retrieve the user request and sends it to the J2EE container. The J2EE container processes the request, invokes the required application components and produces response that is sent back to the client by the Web container.

4.2.1. Web container. Web servers can be constructed following different approaches, depending on the mechanism used to manage client connections and on

the underlying I/O capabilities available in the execution environment. Although the most common implementation for a concurrent web server is the multithreaded architecture, other implementations (such as event-driven architectures) are possible. In this section we will describe some common and basic tuning procedures for multithreaded web servers that do not apply necessarily to other server architectures.

The core of a multithreaded web server is composed of a pool of worker processes or threads, used to service the requests received from a web client, and a thread used to accept new incoming connection requests on the system and to attach each new connection to a free worker thread of the pool. In contrast, an event-driven web server core operates performing non-blocking I/O operations on the sockets where web clients are connected. This feature makes possible to use just one thread to service several clients concurrently. An evaluation of the performance potential of these two architectures is discussed in [9].

Generally, the most basic parameters that can be modified in any multithreaded web container are the size of the pool of threads and the connection timeout value. These factors can have an enormous impact on the web server performance but must be contextualized with the particularities presented by the version of the HTTP protocol chosen for the server, which may vary the way how established connections are managed.

The version 1.0 of the HTTP protocol (HTTP/1.0) states that the client establishes a new TCP connection for each new request that has to be sent to the web server, and that the connection is closed by the server after the corresponding reply is sent back to the client. One single resource is requested with each request. With this approach, several new TCP connections are established between the client and the server for each new request issued, because HTML pages usually include a number of embedded references that must be requested independently through new TCP connections. With the arrival of the HTTP/1.1 protocol (and previously with some extensions to the 1.0 version), the interaction between web client and servers changes slightly. The new version of the protocol states not only that connections are persistent (so they can be reused to process several requests) but also that several requests can be sent together (as a pipeline) to the server. As it can be deduced, using HTTP/1.1 instead of HTTP/1.0 can result in a dramatic reduction in the number of TCP connections active in the server and the consequent reduction on the system resources required to attend several clients simultaneously.

The first of the parameters of a multithreaded web sever that must be adapted to the expected workload is the size of the pool of threads, because it limits the amount of requests that can be attended simultaneously. Remember that each thread of the pool (namely a worker thread) is associated with a TCP connection until it is closed or lost. So, the maximum number of simultaneously attended connections is limited by the number of available worker threads. It is not a real problem for HTTP/1.0 connection because each connection is closed after one request is served, but it represents a serious problem for HTTP/1.1 connections as far as they are persistent and can be kept alive after a request is fully processed and served. In this situation, if as many connections as threads are in the server pool are established, all the threads are busy and the server begins to reject new incoming connections from other web clients which results in a server performance saturation. Therefore, it is recommended to adjust the thread pool size to the expected sustained number of concurrent clients (and leaving a certain superior margin to absorb eventual surges that can appear in the load intensity).

The second basic parameter that should be adapted in the web server for a given workload is the connection timeout, which defines the amount of time an active persistence connection can remain inactive (no requests are sent) without being closed and freed its associated worker thread. This parameter must be defined taking into consideration the expected number of concurrent clients connected to the server, the number of threads available in the thread pool and the expected workload intensity (analyzed in several studies such as in [8]). It is remarkable that many web servers automatically reduce their connection timeout setting as the number of free worker threads in the pool decreases, allowing them to be freed more quickly.

Several guides exist for each commercial web server (or web container) that can help tuning the server, but the basic parameters described above will define the major numbers in the performance under a given workload. For a more detailed study of the internal performance of a commercial web server, such as apache, the reader can refer to [25].

4.2.2. The middleware container. A middleware container is a framework that makes possible the use of light components to create complex pieces of software with relatively low development costs. This is the case of the J2EE and the .NET environments, that offer a rich amount of facilities to isolate user-developed software components from the complexity implicit in the creation of interoperable, accessible and scalable services.

Web containers and middleware containers can run in the same system or in different interconnected machines. If they run in the same machine, they can be part of the same process or work as independent processes. Depending on which one of the named configurations is chosen, the environment must be tuned in a different way.

If the middleware container and the web container run in the same process, the worker threads of the web container (that process client requests) are also the threads that execute the code of the middleware container. Therefore, they go through the code of the middleware container, invoke the required software component, and send back a HTTP reply to the web client. If the middleware container runs as an independent process from the web container, they must communicate using a network. This network will be physical if they run on different machines or a loopback network if they are independent processes running in the same machine. In any case, the communication between them must be done using some type of software adaptors that are used to make the web container and the middleware container talk the same language. At this point, the web container becomes a client application for the middleware container, which must be prepared to accept and process incoming requests and it must be tuned using the same techniques already discussed in section 4.2.1.

Independently of the structural configuration of the system (two containers in the same process or in independent ones, running in the same machine or in remote systems), there is a general constrain that must be taken into consideration: the code of the middleware container is usually full of critical regions and synchronization points that lead to a general slowdown of the system if many requests are serviced concurrently. This characteristic of most middleware containers recommends the use of some kind of admission control for the requests that go from the web container to the middleware container. When the most usual configuration is used in the middleware container (a thread pool is created to accept the requests forwarded by the web container) the recommended configuration is to reduce the thread pool size with respect to the size of the web container thread pool. This causes a resource competition in the web container (which holds many web client connections issuing HTTP requests

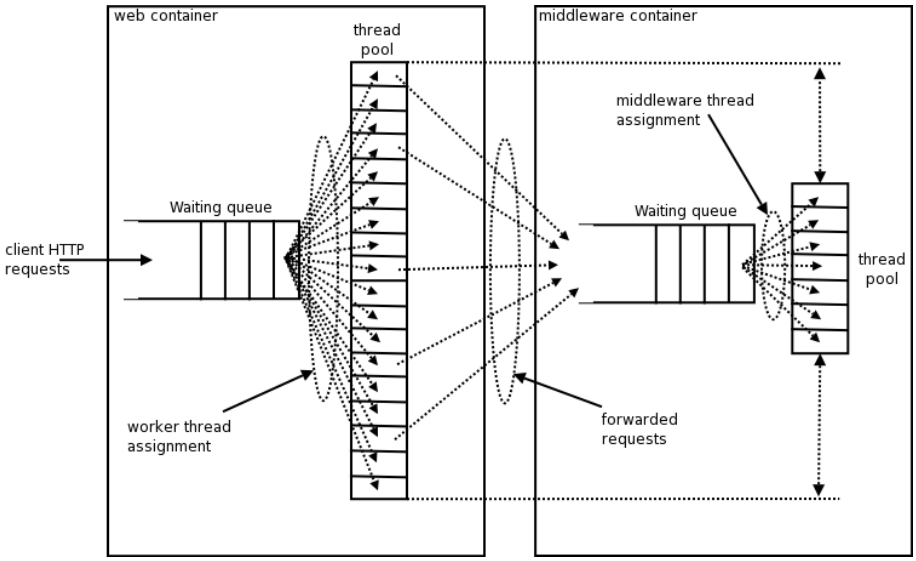

FIGURE 7. Pool size correspondence between web and middleware container

but disposes of few connections to the middleware container to forward the requests) but improves the overall performance of the middleware container and, in extension, of the whole system. A simple diagram of this relation is shown in figure 7.

All containers, web or middleware, are usually accompanied of a number of configuration guides that can help system administrators in the task of tuning the execution environment. An example of these guides can be found in [26], where a full set of configuration tips for a performance optimized configuration of the WebSphere Application Server are detailed.

4.3. Java virtual machine. Middleware containers are great resource consumers, specially with respect to system memory. They are usually object oriented platforms, and in many cases based on virtual execution environments, namely Virtual Machines. In a J2EE environment, the execution environment is provided by the Java Virtual Machine (JVM), which interprets the language and offers a virtual execution environment to Java applications.

The memory parameters to be configured for most Java Virtual Machines are related with the size of the memory heap that is made available to applications as their virtual execution environment. Usually, the maximum and minimum size for the heap is configurable as well as its initial size. Using lower heap sizes will result in lower physical memory consumption but also in a higher amount of garbage collection. Setting the size of the heap according with the expected (or experimentally determined) amount of memory required by an application can save a lot processor time spent in garbage collection, as well as reserving initially the amount of required memory for an application instead of letting the JVM make it grow according to the application requirements can also prevent a number of garbage collector interventions and help increasing the overall performance of the system.

4.4. Operating system and network. A Grid middleware based on the Web Services technology and supported by a middleware container is considered an I/O intensive application. It is in charge of dealing with several network connections

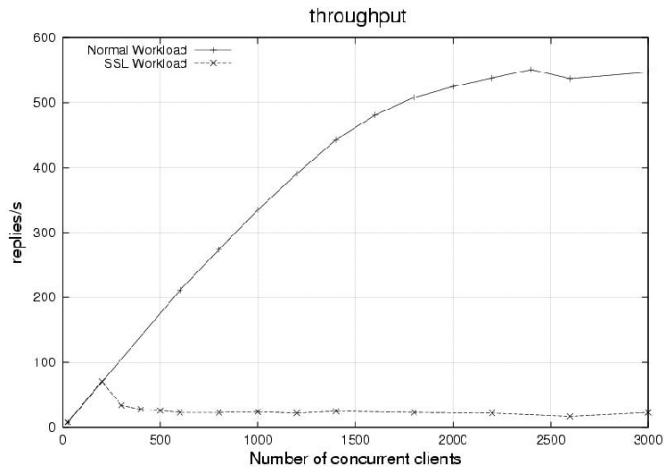

FIGURE 8. Performance impact of SSL on a web server throughput

and the cause of an intensive filesystem activity. This means that the underlying operating system must be successfully configured to avoid the I/O subsystem of being a performance bottleneck. Additionally, grid middleware systems depend strongly on the performance of the network subsystem that interconnects the system with the other machines of the grid.

Tuning an operating system to make applications accessible to a big number of clients and do it efficiently is a hard task. There are many parameters that must be tuned in the system, such as the maximum number of file descriptors that the system can store in its internal tables, the maximum number of threads and processes that are supported in the system or parameters related to the network configuration.

A good reference for the Linux operating system can be found in [30], where many of these topics are discussed. Other proprietary systems (or Linux distributions) may be sold with system tuning guides, but the principles of the tuning techniques are widely described in the suggested reference.

4.5. Security implications. An important part of the communication that takes place in a grid environment uses common encryption techniques to guarantee the privacity of the exchanged information. The most common encryption technique for web transactions is the use of the Secure Socket Layer[21] (SSL) protocol. The use of this encryption protocol can result in a big performance impact for a middleware container because of the amount of computation time required by the encryption/decryption algorithms.

SSL is based on cryptographic techniques and, for this reason, it becomes a computational demanding process. As it is studied in [10], the performance of a middleware container as well as the performance of an associated web container can be sensibly decreased when the system spends more time in the SSL protocol handshaking stage than doing real work for the client requests. An example of this situation can be seen in figure 8, extracted from [10]. It can be seen that the performance obtained when running an application with or without the use of SSL can be very different, comparing the throughput of the system when a certain load (equivalent to

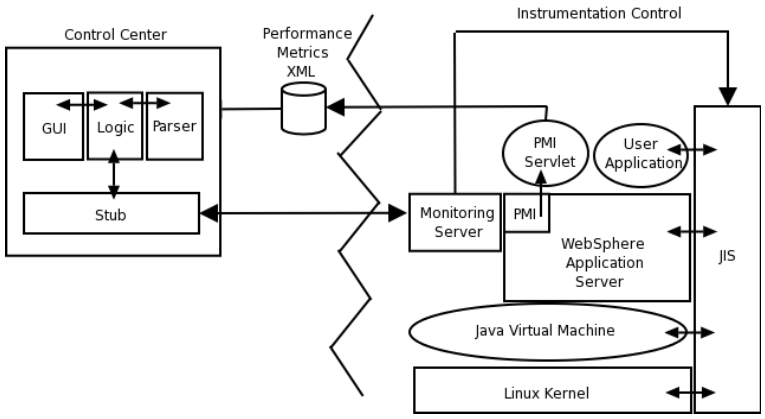

FIGURE 9. WAS Control Center architecture

a number of concurrent connected clients) is supported by the web container.

5. Improving the J2EE container performance. The J2EE container performance is an active research area that is continuously producing new ideas and techniques to obtain a higher throughput from the existing application servers. Some of the research efforts are focused on the creation of new programming paradigms, others on the dynamic reconfiguration of the middleware containers to reduce the middleware overheads and some others work in the improvement in the relation between the application server and the external systems that work cooperatively with it, like database servers and legacy systems.

In the area of the J2EE container improvement, many research groups work under the premises of the Autonomic Computing[28], which is a hot topic promoted by IBM. The idea behind these two words is the creation of systems that can work in an autonomic fashion, being able to diagnose their problems in real time, to solve them dynamically and to tune their configurations automatically to give the best of themselves. At the same time, the topic is divided into several subtopics, covering each one a partial area defined inside of the autonomic computing. Some examples of these subtopics are self-managing systems, self-tuning systems and self-healing systems.

Nowadays, there are no fully autonomic systems operating yet but there are many efforts to understand and improve the complex behavior of J2EE application servers. Some of these efforts try to create cooperative execution environments that exchange information vertically between levels in order to obtain a better system exploitation via the use of resource management techniques. This is the case of the eDragon[1] research group, which works towards the creation of a cooperative environment between the application server and the operating system, and which already has a background knowledge in the application of these techniques to numerical Java applications, as it is discussed in [24]. The first step towards the creation of a definitive resource management environment is to carry out a detailed study of the performance of J2EE application servers under several circumstances in order to characterize it. This work is partially covered in [23] and in [10], and is helped by the analysis tools detailed in [15] (already discussed in section 3). Additionally, the group is working in parallel in the creation of the appropriate self-managing environment for J2EE application servers, as it is discussed in [14] (a simple diagram of the architecture proposed on it is shown in figure 9), based on the use of a portable remote agent that monitors the

performance of the server and sends the necessary control messages to the server in order to adjust its configuration to obtain the higher possible performance.

6. Summary. The grid infrastructure is composed of several nodes that can offer computational services as well as brokering services. If the current architectural trends are maintained, in a future many of this nodes will run grid middlewares that will rely on execution frameworks such as J2EE or the .NET platform, and that will be based on the Web Services technologies in order to become available to the grid users. The performance impact that a grid middleware can incur to the global grid will depend on the granularity of the jobs that are distributed across the grid nodes, but in most circumstances it can result in a significant grid performance degradation if the execution environment of each node is not tuned with accuracy.

In this chapter we have seen that the extreme complexity of the execution stack of a web services-enabled grid middleware introduces many parameters to be tuned on each node, which implies that reaching the maximum overall performance is not a simple task. Refining the configuration of a grid node forces the system maintainer to conceive the grid middleware execution stack (assuming that it is constituted of all the levels ranging from the operating system to the core grid services, according to figure 1) as a set of cooperative software layers, that must be adjusted each one according to the configuration of the others. We also have discussed how a complete performance analysis of a grid middleware must rely on tools that can cover all the levels involved on its execution in order to avoid losing information about the behavior of the application. Finally, we have studied which are some of the active research areas that try to increase the maximum performance that J2EE middlewares will offer in the future.

REFERENCES

[1] *eDragon Research Group. See http://www.ciri.upc.es/eDragon.*

[2] *Extensible Markup Language (XML). Available from http://www.w3.org/XML/.*

[3] *The Globus Alliance. http://www.globus.org.*

[4] *Jakarta Project. Apache Software Foundation. See http://jakarta.apache.org/.*

[5] *Open Grid Services Infrastructure Working Group (OGSI-WG). http://forge.gridforum.org/projects/ogsi-wg.*

[6] *XML Binary Characterization Working Group. See http://www.w3.org/XML/Binary/.*

[7] P. Asadzadeh, R. Buyya, C. L. Kei, D. Nayara, and S. Venugopal, *Global grids and software toolkits: A study of four grid middleware technologies*, Tech. Report GRIDS-TR-2004-4, Grid Computing and Distributed Systems Laboratory, University of Melbourne, 2004.

[8] P. Barford and M. Crovella, *Generating representative web workloads for network and server performance evaluation*, in Proceedings of the ACM SIGMETRICS'98, Madison, Wisconsin, USA, 1998, pp. 151–160.

[9] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé, *Evaluating the scalability of java event-driven web servers*, in 2004 International Conference on Parallel Processing (ICPP'04), Montreal, Canada, 2004, pp. 134–142.

[10] V. Beltran, J. Guitart, D. Carrera, J. Torres, E. Ayguadé, and J. Labarta, *Performance impact of using ssl on dynamic web applications*, in XV Jornadas de Paralelismo, Almería, Spain, 2004, pp. 471–476.

[11] Borland Software Corporation, *OptimizeIt Enterprise Suite. See http://www.borland.com/optimizeit/.*

[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn., H. F. Nielsen, S. Thatte, and D. Winer, *Simple object access protocol (soap) 1.1*, w3c note, W3C, May 8th 2000.

[13] M. Cai, S. Ghandeharizadeh, R. Schmidt, and S. Song, *A comparison of alternative encoding mechanisms for web services*, in Proceedings of the 13th International Conference on Database and Expert Systems Applications, Aix en Provence, France, 2002, pp. 93–102.

[14] D. Carrera, D. Garca, J. Torres, E. Ayguadé, and J. Labarta, *Was control center: An autonomic performance-triggered tracing environment for websphere*, in Proceedings of 13th Euromicro Conference on Parallel, Distributed and Network-based Processing, Lugano, Switzerland, 2005, pp. 26–32.

[15] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, and J. Labarta, *Complete instrumentation requirements for performance analysis of web based technologies*, in Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, Texas, USA, 2003, pp. 166–175.

[16] R. Chinnici, M. Gudgin, J.-J. Moreau, and S. Weerawarana, *Web services description language (wsdl) version 1.2*, tech. report, W3C, July 9th 2002.

[17] K. Chiu, M. Govindaraju, and R. Bramley, *Investigating the limits of soap performance for scientific computing*, in Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing (HPDC'02), Edinburgh, Scotland, 2002, p. 246.

[18] ej-technologies, *Jprofiler. See http://www.ej-technologies.com/products/jprofiler/overview.html.*

[19] European Center for Parallelism of Barcelona, *Paraver. See http://www.cepba.upc.es/paraver.*

[20] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, *The physiology of the grid: An open grid services architecture for distributed systems integration*, 2002.

[21] A. O. Freier, P. Karlton, and P. C. Kocher, *The ssl protocol. version 3.0. http://wp.netscape.com/eng/ssl3/ssl-toc.html.*

[22] B. D. Goodman, *Squeezing SOAP: GZIP enabling Apache Axis, developerWorks, March 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-sqzsoap.html.*

[23] J. Guitart, D. Carrera, J. Torres, E. Ayguadé, and J. Labarta, *Tuning dynamic web applications using fine-grain analysis*, in Proceedings of 13th Euromicro Conference on Parallel, Distributed and Network-based Processing, Lugano, Switzerland, 2005, pp. 84–91.

[24] J. Guitart, X. Martorell, J. Torres, and E. Ayguadé, *Application/kernel cooperation towards the efficient execution of shared-memory parallel java codes*, in 2003 International Parallel and Distributed Processing Symposium (IPDPS-2003), Nice, France, 2003, p. 38a.

[25] Y. Hu, A. Nanda, and Q. Yang, *Measurement, analysis and performance improvement of the apache web server*, in 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99), Phoenix/Scottsdale, Arizona, USA, 1999, pp. 261–267.

[26] IBM Corporation, *IBM WebSphere Application Server, Advanced Edition. Tuning Guide.*

[27] ———, *WebSphere Studio Application Developer. See http://www-306.ibm.com/software/awdtools/studioappdev/.*

[28] IBM Research, *Autonomic computing. See http://www.research.ibm.com/autonomic.*

[29] Jakarta Project. Apache Software Foundation, *Tomcat. See http://jakarta.apache.org/tomcat.*

[30] D. Kegel, *The C10K problem, http://www.kegel.com/c10k.html*, 2003.

[31] Microsoft, *Microsoft .NET technology. See http://www.microsoft.com/net/.*

[32] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley, *Network performance effects of http/1.1, css1, and png*, in Proceedings of the ACM SIGCOMM '97, Cannes, French Riviera, France, 1997, pp. 155–166.

[33] OperSys Inc., *Linux Trace Toolkit (LTT). See http://www.opersys.com/relayfs/ltt-on-relayfs.html.*

[34] Quest Software, *JProbe. See http://www.quest.com/jprobe/.*

[35] ———, *PerformaSure. See http://www.quest.com/performasure.*

[36] D. M. Sosnoski, *Improve XML transport performance, Part 1, developerWorks, June 2004. http://www-106.ibm.com/developerworks/xml/library/x-trans1/.*

[37] D. M. Sosnoski, *Improve XML transport performance, Part 2, developerWorks, June 2004. http://www-106.ibm.com/developerworks/xml/library/x-trans2/.*

[38] Sun Microsystems, Inc., *Fast Infoset. See http://java.sun.com/developer/technicalArticles/xml/fastinfoset.*

[39] ———, *Fast Web Services. See http://java.sun.com/developer/technicalArticles/WebServices/fastWS.*

[40] ———, *Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee.*

[41] ———, *Java 2 Platform, Standard Edition (J2SE). See http://java.sun.com/j2se.*

[42] ———, *Java servlets technology. see http://java.sun.com/products/servlet/.*

[43] The Apache Software Foundation, *Apache Axis. See http://ws.apache.org/axis/.*

[44] The Globus Alliance, *Globus Toolkit. http://www.globus.org/toolkit.*

[45] D. Viswanathan and S. Liang, *Java virtal machine profiler interface*, in IBM Systems Journal, 39(1):82-95, 2000.

[46] G. Wasson, N. Beekwilder, M. Morgan, and H. Humphrey, *Ogsi.net: Ogsi-compliance on the .net framework*, in Proceedings of the 2004 IEEE International Symposium on Cluster

Computing and the Grid (ccGrid'04), Chicago, Illinois, USA, 2004, pp. 648–655.
[47] World Wide Web Consortium (W3C), *Web Services Activity. http://www.w3.org/2002/ws/*.