The Grid Architectural Pattern: Leveraging Distributed Processing Capabilities

Raphael Y. de Camargo, Andrei Goldchleger, Mrcio Carneiro, and Fabio Kon

Department of Computer Science
University of So Paulo
{rcamargo,andgold,carneiro,kon}@ime.usp.br

May, 2005

Grid Middleware

The Grid Middleware pattern describes the software infrastructure necessary to allow the sharing of distributed and potentially heterogeneous computational resources for execution of applications. The middleware deals with several areas, such as resource management, scheduling, and security in an efficient and transparent manner. This pattern addresses both the architecture and implementation of the middleware.

Example

Weather forecasting is a typical computationally intensive problem. Briefly describing, data regarding the area subject to forecasting is split into smaller pieces, each one corresponding to a fraction of the total area. Each fragment is then assigned to a computing resource, typically a node on a cluster, or a processor in a parallel machine. During the computation, nodes need to exchange data, since the forecasting in each of the fragments is influenced by its neighbors. After several hours of processing, the results of the computation are expected to reflect the weather on the given area for a certain period, a few days for example. Figure 1 shows the results of a simulation where the total area was divided in 16 fragments.

There are several other applications that can be broken into smaller pieces and require large amounts of computational power, such as financial market simulation, image processing, three-dimensional image generation, bioinformatics, signal analysis, etc.

In order to perform computations such as the ones described, one typically uses dedicated infrastructures, such as parallel machines or dedicated clusters. Institutions usually have a limited

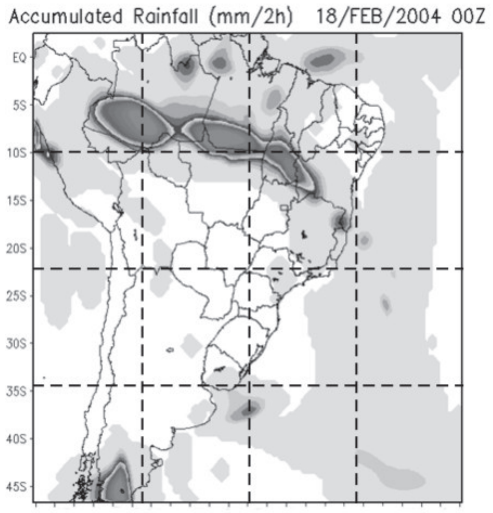

Figure 1: Weather forecasting.

amount of these resources, which are disputed by many users in need of processing power. At the same time, it is very likely that in these same institutions there are hundreds of workstations laying idle for most of the time. If these workstations were to be used for processing during these idle periods, they would multiply the processing power available to the users.

The term Grid computing [FK03, BFH03] is used to denote the coordinated sharing of distributed resources, providing ubiquitous access to remote resources and, consequently, increasing the amount of available computing power. Grid computing requires a sophisticated middleware to coordinate the resource sharing process, dealing with resource management, scheduling, and security in an efficient and transparent manner.

Grid Computing is a vast and active research area which encompass various aspects of distributed computing such as seamless access to distributed data (Data Grids), and collaborative distributed environments (Service Grids). This pattern focuses on Grids for running computationally intensive applications (Computational Grids), focusing on Grids that leverage the idle capacity of commodity workstations.

Context

Sharing of computational resources in heterogeneous distributed systems in order to improve the availability of computing resources for the execution of computationally intensive applications.

Problem

Using the idle periods of available computing resources can greatly increase the amount of computing power available to users of an organization. A software infrastructure (Middleware) that allows the use of these shared computing resources must address aspects such as application deployment, distributed scheduling, collection of execution results, fault-tolerance, and security. The following ***forces*** must be considered when developing such an infrastructure:

- Grids can be composed of thousands of machines. The design and implementation of the Grid Middleware should be scalable.
- The Grid will encompass heterogeneous resources, with different computing architectures and operating systems. The middleware system must account for this diversity.
- By default, the Grid Middleware underlying mechanisms should be transparent for end users. The system can optionally allow the user to specify some aspects of application execution, such as manually selecting the machines that will execute the application.
- Different administrative domains have custom policies for resource sharing and usage. The middleware should be flexible to allow these different policies to be enforced.
- In order to encompass a large amount of resource providing machines, the middleware must be easy to deploy, without requiring extensive reconfiguration on existing systems.
- Security must be provided for both resource owners and users submitting applications. Machines sharing their resources need protection against malicious code. Grid users should have some guarantees regarding data confidentiality and integrity.
- A large number of existing parallel applications exist that could benefit from the Grid. The middleware must provide grid-enabled libraries supporting standard parallel programming models, allowing for easy application migration to the Grid environment.

Solution

Use a middleware infrastructure to manage distributed and potentially heterogeneous resources, allowing users to access these resources for the execution of computationally intensive applications.

A user interested in executing applications submit a request via an *access agent*. The applications can be either regular applications consisting of a single process, or parallel applications. The *scheduling service* receives the execution request, checks the identity of the user who submitted the application, and uses a *resource management service* to discover which machines (nodes) have available resources to execute the application. These nodes, which must run the *resource provision service*, are called *resource providers*. If there are nodes with free resources, the scheduler sends the jobs[1] to the selected nodes. Otherwise, the request waits in an execution queue. When the execution is finished, the results are returned to the user.

[1]In this work we use the term *job* to refer to each of the processes of an application.

Structure

The structure of the Grid Middleware pattern is composed of five main components. Figure 2 shows the interactions between these components.

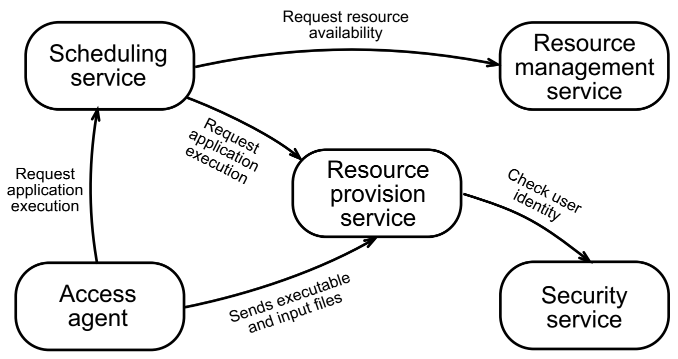
Scheduling service
Request resource availability
Resource management service
Request application execution
Request application execution
Resource provision service
Check user identity
Access agent
Sends executable and input files
Security service

Figure 2: Relationships between the Grid Middleware modules.

- The *access agent* is the primary access point for users that interact with the Grid. It runs on each node from which application execution requests will be submitted. Besides the submission of execution requests, it allows the user to specify application requirements, monitor executions, and collect execution results.
- The *resource provision service* runs on each machine that exports its resources to the Grid. It is responsible for servicing execution requests by retrieving application code, starting application execution, reporting errors on application execution and returning application results. This service is also responsible for managing local resource usage and providing information about resource availability.
- The *resource management service* is responsible for monitoring the state of shared resources and responding to resource requesting, by matching the requests with resource offerings. It maintains a list of which resource providers are currently available and the state of their resources, such as processor usage and free memory. This service can also detects resource provider failures, and notifies the access agent when an execution fails due to a resource failure.
- The *scheduling service* schedules execution of applications on available shared resources. It receives application execution requests, obtains available resources with the resource management service, and then determines which application will execute on each resource.
- The *security service* is responsible for three major tasks: (1) protecting shared resources, so that a node sharing its resources with the Grid does not suffer the effects of a rogue application, (2) user authentication, so that application ownership can be established, enabling

relations of trust and accountability, and (3) securing Grid communications, providing information confidentiality and integrity.

Dynamics

We illustrate the behavior of a possible implementation of the Grid Middleware pattern when an access agent requests an application execution (shown in Figure 3). We consider that no errors occur during the execution of the application.

- The access agent sends an execution request to the scheduling service, possibly with some information about the required resources.
- The scheduling service queries the resource monitoring service for nodes meeting the request requirements. A list containing the location of nodes satisfying the query is returned.
- The scheduling service determine if it is possible to execute the application in the available nodes. If the execution is possible, the request is sent to the selected resource providers. Otherwise, it queues the request for later execution.
- The resource providers download the application code from the access agent that submitted the execution request.
- The resource providers use the security service to verify the identity and permissions of the code owner.
- The resource providers executes the application. When the execution is finished, the results are sent back to the access agent.

Implementation

Implementing the Grid Middleware pattern is a complex process. In this section we present some guidelines for the implementation and deployment of the pattern.

- *Communication infrastructure:* is necessary in order to allow Grid Middleware components to locate each other and exchange information. It should provide a reliable communication mechanism, such as remote procedure calls (RPC) or asynchronous messaging, and work on all platforms where Grid Middleware components will be installed.

  It is easier to use an already existing communication infrastructure that satisfy the Grid requirements. To permit the usage of heterogeneous machines, the infrastructure should be language and platform independent, for example CORBA [Obj02a] and Web Services. They permit decoupling the module interfaces from their implementations, allowing multiple interchangeable implementations that support different architectures. Both are implementations of the Broker [BMR$^+$96] pattern.

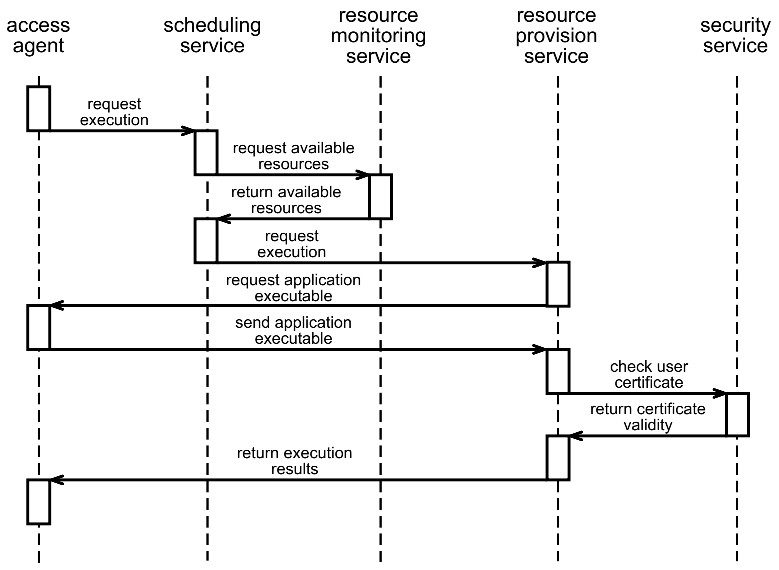
access agent
scheduling service
resource monitoring service
resource provision service
security service
request execution
request available resources
return available resources
request execution
request application executable
send application executable
check user certificate
return certificate validity
return execution results

Figure 3: **Dynamics** of a successful execution request.

Another possibility is to develop a new infrastructure from scratch. Here, it is necessary to deal with several aspects of the infrastructure, such as communication reliability, event dispatching, machine heterogeneity, and component interfaces. Several patterns can be used for the construction of this communication infrastructure [HW03].

- *Access agent:* is typically implemented as a proxy that receives user requests and forwards it to others Grid Middleware components. It must provide a well defined API that allows one to make execution requests, specify application requirements and preferences, monitor application execution, and collect executions results. A friendly interface to the access agent should be provided to permit users to interact with the Grid. This user interface can be implemented as a shell-like interface, for example using UNIX-like commands, a graphical interface, or a Web portal.

  During an execution request, application binary is usually sent by the access agent directly to the resource providers. Input and output files can be treated using different approaches. A first possibility is to allow the access agent to send the input files with the request and, when the execution finishes, download the output files directly from the resource providers. Another option is to define a data repository in the Grid, uploading the input and output files to this repository. Finally, it is possible to link the Grid applications with a remote I/O library, allowing reading and writing remotely to files in the machine from where the execution request was performed. These approaches are used in different Grid Middleware pattern implementations.

  In the execution request, application requirements and preferences can also be specified. Typical constraints are the architecture on which the binary was compiled and the amount of RAM necessary to run it. Common preferences are desired processor speed and network bandwidth. A constraint language that allows the specification of the requirements and preferences of the application should be provided.

- *Resource provision service:* runs permanently on each of the nodes sharing its resources. This service should have a small memory footprint when deployed on shared machines, in order to assure that resource owners do not perceive a degradation in the quality of service provided by their machines.

  When this service receives an *application execution request* from the scheduling service, it is necessary to obtain the application binaries and other input files necessary for the execution. This process depends on how these files are uploaded by the access agent. It is then necessary to create an execution environment for the application, what can be performed by creating a separate directory for each execution, and launching a new process for execution on that directory. After the execution finishes, the output files from the execution must be returned. In the case of abnormal terminations, partial results and error messages should be returned to the access agent in order to allow the user to analyze the cause of the failure.

  This service is also responsible for sending periodic *resource usage information* to the resource monitoring service. A shorter update interval leads to higher network traffic, while longer intervals can lead to stale information. A common approach is to send updates only when a substantial change occurs. Keep alive messages can be used to monitor if the resource provider is active.

  Management of local resources can be performed either by the operating system scheduler, or by a custom scheduler. Using a custom scheduler, it is possible to better control resource utilization by Grid applications, but depending on the operating system may have to run at system level. Support for resource reservation permits better QoS guarantees, but usually requires a custom local scheduler.

  Finally, the resource provision service should provide an interface that allows the resource owner to specify *constraints on resource sharing*. It should provide the possibility of specifying restrictions such as which users can access the resources, the period of the day on which the resources can be shared, and the amount of resources to be shared.

- *Resource management service:* needs to provide interfaces for both queries and updates regarding the availability of resources from resource providers. It maintains a list containing the registered resource providers, their characteristics, and information about the currently available resources. A query language that allows the specification of application requisites and preferences should be supported, in order to perform the matchmaking between resource requests and offers.

  In order to keep the information up to date, this service receives periodic updates from resource providers concerning their resource availability status. It is possible to use push, pull, or push/pull approaches for the periodic update. These updates also allow the monitoring service to detect when resource providers are not reachable, which can happen due to hardware, software, or network failures. When it is detected that a node is unreachable, this service should inform the scheduler or access agent about the failure on the node and the processes that were executing on that node, so that these processes can be reinitialized in another resource provider.

When dealing with large numbers of machines spanning distinct administrative domains and/or geographically dispersed, it is necessary to treat some extra issues, such as scalability and different organizational policies. A centralized solution can become a bottleneck when there are thousands of geographically dispersed machines. Besides, it involves a single point of failure and concentrates management under a single administration. A distributed architecture for this service can be used, where each server maintains the status of part of the resources. This prevent single servers from having complete knowledge about the network status and is usually more tolerant to system and network failures. It is also usually easier for each organization to manage and deploy its own resource management service, applying the desired policies for their shared resources.

The storage of dynamic resource availability information is easier to be implemented using existing solutions, such as the CORBA Trading Service [Obj00] or LDAP [YHK95]. Another possibility is to implement the service using a database, providing a custom front-end for queries and updates.

- *Scheduling service:* is a particularly difficult task for Grid computing. There are many types of application with different requisites, what makes one-fit-all algorithms not viable. Another complications is that the scheduler typically does not have reliable information about the resources status, and does not have full control over these resources. Finally, the heterogeneity of the resources makes scheduling an even harder task.

  A possible solution is to implement the scheduler in the access agent, which asks the resource manager for available resources, and then schedules the applications submitted to that access agent in the returned resources. This allows different clients to employ different scheduling strategies, including the definition of user priorities.

  Another possibility is to implement the scheduling service as a separate component containing a default scheduler, and domain specific schedulers. Application are then allowed to choose one of these schedulers or implement a custom one. The Strategy pattern [GHJV95] can be used to allow the scheduler to employ these different scheduling strategies.

  In the case of multiple resource managers where each manages only part of the resources, the scheduler queries the resource manager from its cluster for available resources. If resources are not found there, the scheduler queries other resource managers.

  The service should signal an error to the user when the resources required by the application could not be found. The error message may provide some information about the available resources, so that the user might relax the application requirements to meet the available resources.

- *Security service:* resource providers can be protected by limiting system privileges for Grid applications. This can be accomplished by presenting a restricted execution environment for the Grid applications, usually referred as a sandbox. It can be implemented by intercepting system calls or using a virtual machine, as in a Java environment. Typical restrictions include disallowing the application to fork new processes, access some peripherals such as printers, and obtain unrestricted access to the filesystem. The authorization mechanism should provide a method to allow resource owners to specify these security restrictions, which can be different depending on the owner of the application to be executed. This can be implemented by using access control lists (ACL), which are composed by lists of actions that are allowed or denied for a user. Since the resource provider protection is enforced by the operating system, a mapping between Grid users and users on the local machines becomes a necessity.

  A requirement for the user authentication mechanism is to provide a single sign-on facility for Grid users. The standard way of authenticating an user is by use of passwords. After checking the password, the authentication mechanism can generate a certificate that identifies that user. Credentials can be verified either by providing the public key of the certificate owner, or by intermediation of a third-party certificate issuer server.

  Communication security incurs an execution overhead, specially for parallel application that exchange large amounts of data. For applications that do not need information confidentiality and/or integrity, it should be possible to disable this mechanism. The information regarding the need for communication security for an application could be included in the application preferences defined before application submission.

  The implementation of both user authentication and data encryption can be greatly simplified by leveraging existing security libraries. GSS (Generic Security System) [Lin93] is a standard API for managing security; it does not provide the security mechanisms itself, but can be used on top of other security methods, such as Kerberos [NT94]. CorbaSEC [Obj02b] is another alternative, but it can be used only if the communication infrastructure is also based on CORBA.

- *Libraries for parallel computation:* in order to allow the execution of parallel applications, it is necessary to implement Grid-specific libraries for different parallel application models. It is important to include well-known APIs for parallel programming, such as MPI [For93], BSPlib [Val90], and PVM [Sun90], since it permits easier migration of already existing software. The implementation of these libraries can be done either from scratch or by modifying existing libraries to use the Grid Middleware communication and resource management infrastructures.

  Another approach is to implement libraries that allows the development of programs following some execution model. An example is the master-worker (WS) model, where a master process is responsible for the coordination of several independent worker processes. Examples of applications for this model include parameter searching, signal processing, and image generation.

- *Opportunistic Computing:* when using shared workstations to run Grid applications, it is necessary to preserve the quality of service (QoS) for the workstation owner, who should not perceive a substantial drop in performance. When the owner of an idle workstation decides

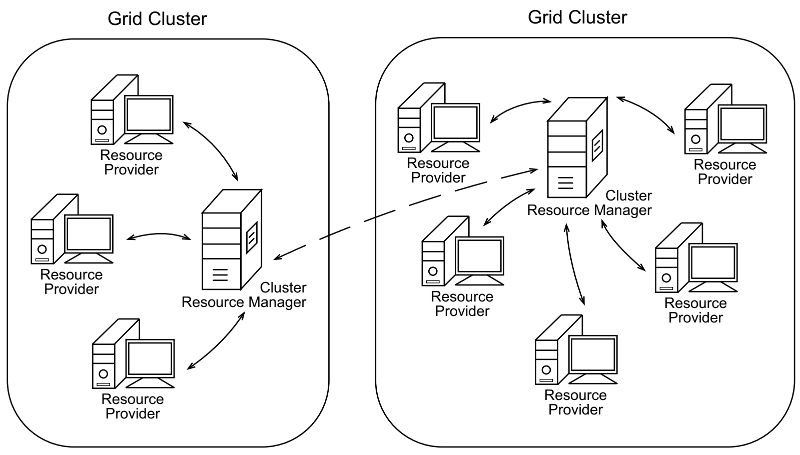

Figure 4: A typical Grid Middleware deployment.

to return to the machine, the processes from Grid applications running on that machine must be killed or suspended.

Since in this situation the rate on which processes are killed is much higher then in the case where resources are dedicated to processes from the Grid, it is important to be able to restart the execution of these processes from an intermediate state, not from its beginning. In the case of parallel applications this is particularly important because the failure of any of the application processes can require all the remaining processes to also be reinitialized. A mechanism that allows the reinitialization of the application from an intermediate execution state is *Checkpointing* [EAWJ02]. It consists in periodically saving the application state as a checkpoint, allowing recovering application execution from the last saved checkpoint.

Even using checkpointing, part of the computation performed by the application is always lost. Besides, the application files need to be uploaded to another node. This makes the reinitialization process expensive. In order to reduce the number of these reinitializations, *usage patterns* for the resource providers can be determined by using clustering or time series algorithms on the usage history of the offered resources. This data can be used by the scheduling service in order to select machines with a higher probability of remaining free for a larger period of time.

- *Deployment of the Grid Middleware:* the resource provision service must be started on each machine that will export its resources to the Grid. Each organization then defines policies for the shared resources. Access agents are deployed on machines belonging to users that submit applications for execution.

  If using a distributed resource manager service, a resource manager server is deployed on each administrative and/or geographic domain. This server is responsible for the resource providers from that domain. Security deployment is dependent on the implementation. If using centralized certificate-based approaches, it is necessary to deploy the certificate servers. Figure 4 shows a typical Grid Middleware deployment for two distinct administrative domains.

Known Uses

Globus [FK97, BFH03] is a Grid middleware that provides a collection of services that allows developers to write applications to execute on Globus Grids. Globus is built as a toolkit, allowing incremental addition of functionalities for the development of Grid applications. It implements most of the services described in this pattern. The services **MDS** (Monitoring and Discovery Service) [CFFK01], and **GRAM** (Globus Resource Allocation Manager) [CFK+98] are cooperatively responsible for the resource management and resource provision services. It also includes a security service based on GSS over Kerberos. PVM and MPI parallel programming APIs are available for running parallel application.

Legion [GWF+94, BFH03] is an object-oriented middleware for Grid computing. Legion treats everything as an object, including hosts, users, schedulers, applications, and data. It started as a academic project, and was later transformed in a commercial product. The services are implemented by Legion core objects. *Host* objects are responsible for the resource provision service and for the security of the resource. *Scheduler* objects implement different scheduling strategy. The list of available resources are obtained from *Collections* objects, which contain lists of Host objects. Finally, Legion provides support for the execution of MPI applications.

Condor [LLM88, BFH03] allows the integration of computational resources to create a cluster for the execution of applications. Condor targets at both high-throughput and opportunistic computing. The *Agent* module implements the functionality of the access agent, and is also responsible for the scheduling process. Resource management is performed by the *Matchmaker*, and the resource provision service is provided by the module *Resource*. A Condor Grid can be formed from several Condor pools, which are a group of machines connected to a single Matchmaker. A number of parallel programming libraries are available to Condor, including PVM, MPI, and one based on the master-work model. Finally, applications can benefit from a transparent checkpointing mechanism, but this mechanism is not portable and is not available for parallel and distributed applications.

InteGrade [GKG+04] is a Grid computing system designed as a middleware infrastructure and programming platform for Grid applications. This system also provides most of the services of the Grid Middleware pattern. Its resource management and scheduling services are implemented as a single module, *Global Resource Manager* (**GRM**). The resource provision service is called *Local Resource Manager* (**LRM**) and has a lightweight implementation that imposes a small overhead on shared resources. InteGrade uses CORBA for the communication infrastructure, includes support for portable checkpointing of sequential and BSP applications, and a security service based on GSS over Kerberos.

OurGrid [ACBR03] is a simple Grid system which allows the execution of sequential and Bag-of-Tasks applications. It is based on a peer-to-peer network of resources, and the resource sharing is based on the concept of network of favors, that is, your priority for the use of resources is determined by the ratio of resources utilization time against sharing time. The access agent is implemented by the module *Client*, which is also responsible for the execution scheduling. The resource provision service is implemented by the *Provider* and *Consumer* modules. The resource can be a single machine or a cluster of machines. There is not a resource management service. A Client asks for resources for the Consumer module on its machine, which broadcast a resource request to the other resource providers of the network.

Consequences

The Grid Middleware pattern provides the following **benefits**:

- *Reusability:* once implemented, the Grid Middleware infrastructure works for many applications, since they will use the same services and programming libraries. Moreover, since the system is a middleware that provides an abstraction layer for the development of applications, it can be easily installed as an off-the-shelf middleware on different systems with little modifications on the underlying systems.

- *Encapsulation of heterogeneity:* the Grid Middleware hides the specific details on communications, computer architectures, and operating systems. Consequently, application development is much easier, without need to worry about details of communication and heterogeneous computer environments.

- *Easy application deployment:* the Grid Middleware manages the details of resource allocation, scheduling, security, and application deployment. This facilitates the execution process, since the user does not have to worry about the details of reserving computing resources and deploying the application on various machines.

- *Efficient resource usage:* resource idleness is significantly reduced when using the Grid Middleware. This applies both to workstations and to specialized and expensive resources. During normal operation, they are used by their owners, without degradation in their Quality of Service. But when idle, these resources can be shared to be utilized by applications running on the Grid. The efficient resource usage means that less hardware will have to be purchased and maintained, resulting in a significant reduction in costs.

- *Integration of dispersed resources:* the use of the Grid Middleware eases the integration of dispersed resources, including geographically distant nodes. When these resources are spread across administrative domains, different policies for resource sharing, such as access restrictions, can be implemented for each domain, specially if machines are divided in clusters connected by an inter-cluster architecture.

However, the Grid Middleware pattern has the following **liabilities**:

- *High complexity:* the implementation of the Grid Middleware pattern involves many challenges that still do not have a consolidated solution. Although some of these challenges are applicable for distributed systems in general, the Grid Middleware must employ much more general and comprehensive solutions. Therefore, implementing the Grid Middleware pattern demands a considerable amount of time and effort of talented people, what can incur in a high cost. The investment on must be justified by using the Grid Middleware for many applications and/or environments.

- *Necessity to change applications:* to take advantage of the Grid Middleware, it may be necessary to alter parts of the application. This requires access to the application source code, which is rare on commercial applications. Even when the source code is available, its adaptation for using the Grid Middleware can consume considerable amounts of time and money, for example if the application is written using a parallel programming API not supported by the Grid Middleware.
- *Harder to debug and test applications:* applications submitted for execution on the Grid are harder to test and debug. This happens because the user has no control on where the application processes are executed. In addition, currently available tools for testing and debugging applications in a distributed environment are limited.
- *Security exposures:* Executing foreign code on a resource provider machine is a potential security breach. It is possible to minimize the risk by restricting the execution environment, but it is difficult to fully guarantee the safety of the machine.

See Also

The **Lookup** pattern [KJ04] uses a lookup service for finding and accessing computational resources, and has a structure similar to the Grid Middleware pattern. Its participants correspond to the resource management service, resource provision service, access agent, and resources of the Grid Middleware pattern. The main difference is that the Lookup has a smaller scope, not involving issues such as security, sharing policies, and scheduling.

The **Resource Lifecycle Manager** pattern [KJ04] separates resource usage from resource management, by introducing a Resource Lifecycle Manager. Similarly to the resource provision service, it is responsible for accepting resource usage requests, and managing the resources according to defined policies. But it does not address security concerns necessary for the Grid environment.

The **Broker** pattern [BMR$^+$96] has similarities with the Grid Middleware pattern. Like the Grid Middleware, its goal is to encapsulate several details regarding the implementation of distributed systems, simplifying the development of applications. However, differently from the Grid Middleware, the Broker is recommended for simpler applications, such as business systems based on the client/server architecture. Consequently, it does not require many of the features available in the Grid Middleware, such as the scheduling and resource monitoring services. Since the Broker encapsulates details such as the communication with remote entities, it can be used as a substrate for the implementation of the Grid Middleware pattern. InteGrade uses CORBA, an implementation of the Broker pattern, as the basis for its communication.

The **Master-Slave** pattern [BMR$^+$96] consists of having a central coordinator that distributes tasks for execution on servants and then collect the results. Differently from the Grid Middleware pattern, the pattern also only applies to problems that can be solved by the ‘divide and conquer’ approach. However, this specificity allows a simpler system that can be easily implemented and optimized. This pattern can be used as a model for the development of parallel applications to execute on the Grid.

Acknowledgments

We would like to thank Eugenio Sper de Almeida for providing the weather forecasting image used in our example. Special thanks for Michael Stal, whose innumerable suggestions were invaluable for improving the quality of this pattern. Finally, this work would not be possible without the financial support from the Brazilian Research Council (CNPq).

References

[ACBR03] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, and Paulo Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[BFH03] Fran Berman, Geoffrey Fox, and Tony Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, 2003.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented System Architecture: a System of Patterns*. John Wiley & Sons, 1996.

[CFFK01] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[CFK$^+$98] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of IPPS/SPDP98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[EAWJ02] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, May 2002.

[FK97] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 2(11):115–128, 1997.

[FK03] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 2003.

[For93] MPI Forum. MPI: A Message Passing Interface. In *Supercomputing Conference*, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 4, pages 139–150. Addison-Wesley, 1995.

[GKG$^+$04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, March 2004.

[GWF$^+$94] Andrew S. Grimshaw, Willian A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. A Synopsis of the Legion Project. Technical report, University of Virginia Computer Science Department, June 1994.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[KJ04] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. John Wiley & Sons, 2004.

[Lin93] J. Linn. Generic Security Service Application Program Interface, September 1993. Internet RFC 1508.

[LLM88] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[NT94] B. C. Neuman and T. Tso. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32:33–38, September 1994.

[Obj00] Object Management Group. *Trading Object Service Specification*, June 2000. version 1.0, OMG document formal/00-06-27.

[Obj02a] Object Management Group. *CORBA v3.0 Specification*, July 2002. OMG Document 02-06-33.

[Obj02b] Object Management Group. *Security Service Specification*, March 2002. version 1.0, OMG document formal/02-03-11.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33:103–111, 1990.

[YHK95] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. RFC #1777, March 1995.