Grid: An Architectural Pattern

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

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

June, 2004

Grid

The Grid pattern allows the sharing of distributed and possibly heterogeneous computational resources, such as CPU, memory, and disk storage, in an efficient and transparent manner.

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 PC in a cluster, or a processor in a parallel machine. During the computation, each computational node needs to exchange data with others, since the forecasting in each of the fragments is influenced by neighboring areas. After several hours of processing, the results of the computation are expected to reflect the weather of 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.

In order to perform a computation such as the one described, one typically uses dedicated infrastructures, such as parallel machines or dedicated clusters. Institutions typically have a limited amount of these resources, which are disputed by many users who needs processing power. At the same time, it is very likely that in these same institutions there are hundreds of workstations that remain idle for most of the time. If these workstations could be used for processing during these idle periods, they would multiply the processing power available to the users.

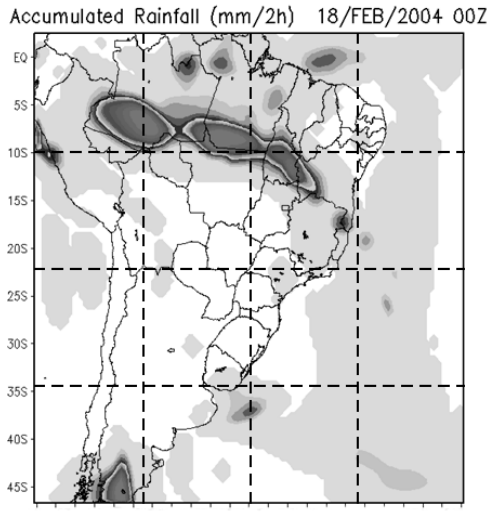

Figure 1: Weather forecasting.

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.

The Problem

Using the idle periods of available computing resources can greatly increase the amount of resources available to users of an organization. However, it is very difficult to do this using traditional off-the-shelf software and operating systems. Many issues have to be treated, such as application deployment, distributed scheduling, collection of results, fault-tolerance, and security. This leads to the necessity of an additional software infrastructure to provide solutions to the above mentioned problems. The following ***forces*** must be considered when developing such an infrastructure:

- It must allow the sharing of computational resources, such as processors, memory, storage, and other kinds of hardware resources, that would otherwise remain idle or underutilized.
- It needs to allow the execution of applications on various computing architectures running different versions of different operating systems.
- It needs to be transparent to the user. The user should not need to know where submitted applications will be executed.
- In the case of shared workstations, it needs to provide quality of service (QoS) to both users

submitting applications and resource owners, by reaching a compromise between their necessities.

- It needs to provide autonomy to users and system administrators, avoiding imposing rigid policies on resource offering.
- It must allow existing applications to be easily adapted to the new context. In particular, support for standard parallel programming APIs should be provided.
- It needs to have a low deployment cost, not requiring extensive reconfigurations on existing systems.

Solution

The Grid pattern can be used to solve this problem. It consists of using a middleware to manage distributed and possibly heterogeneous resources, allowing users to submit applications for execution. These applications can be either single process, in which case none or few changes in the application are required, or parallel applications, which should rely on parallel communication libraries provided by the middleware. These libraries are typically grid-enabled version of traditional parallel libraries, which makes easy to migrate pre-existing applications to the Grid environment.

A user interested in executing applications submit execution requests via an *access agent*. The access agent allows the user to specify execution constraints regarding the application, such as operating system, computer architecture, and memory requirements.

A *scheduling service* receives the execution requests, checks the identity of the user who submitted the application, and uses a *resource monitoring service* to discover which 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 a resource provider receives a job for execution, it creates a new process for that job and puts it in the execution queue of the local machine. When the job execution finishes, the results are then returned to the access agent.

An important point is that all the execution is transparent to the user. After application submission, the Grid is responsible for finding the computational resources, scheduling the application for execution, providing fault-tolerance and returning the results to the user.

- In the case of the weather forecasting application, if it were written using an API supported by the Grid, the user would only have to recompile the application with the libraries provided by the Grid and submit it for execution. When the execution is finished, the user gets the results through the access agent.

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

Communication Infrastructure	
Responsabilities	Interactions
Provides a reliable communication mechanism Provides additional services (e.g. naming service)	Resource Provision Service Resource Monitoring Service Parallel Programming Libraries Security Service Scheduling Service Access Agent

Resource Provision Service	
Responsabilities	Interactions
Services application execution requests Controls local resource usage Provides resource usage information	Communication Infrastructure Resource Monitoring Service Parallel Programming Libraries Security Service Scheduling Service Access Agent

Resource Monitoring Service	
Responsabilities	Interactions
Keeps resource availability and usage information Monitors resource providers	Resource Provision Service Communication Infrastructure Scheduling Service

Security Service	
Responsabilities	Interactions
Protects shared resources Manages user identities PrevidES secure communication channels	Resource Provision Service Communication Infrastructure

Scheduling Service	
Responsabilities	Interactions
Manages Grid resources Schedules application execution requests	Resource Provision Service Communication Infrastructure Resource Monitoring Service Access Agent

Access Agent	
Responsabilities	Interactions
Provides an interface for user interaction with the Grid Allows application execution requests submission and monitoring	Resource Provision Service Communication Infrastructure Scheduling Service

Figure 2: CRC cards describing the Grid modules.

Structure

- The *communication infrastructure* allows the different Grid components to exchange information. It provides a reliable communication mechanism and other services, such as a naming service, to simplify the development of the remaining grid modules.
- The *access agent* is the primary point of access for users to interact with the Grid. It must be executed in each node from which applications will be submitted. Besides the submission of applications, it allows the user to specify application requirements, monitor executions, and collect execution results.
- The *resource provision service* must be executed 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 monitoring service* is responsible for monitoring the state of Grid resources

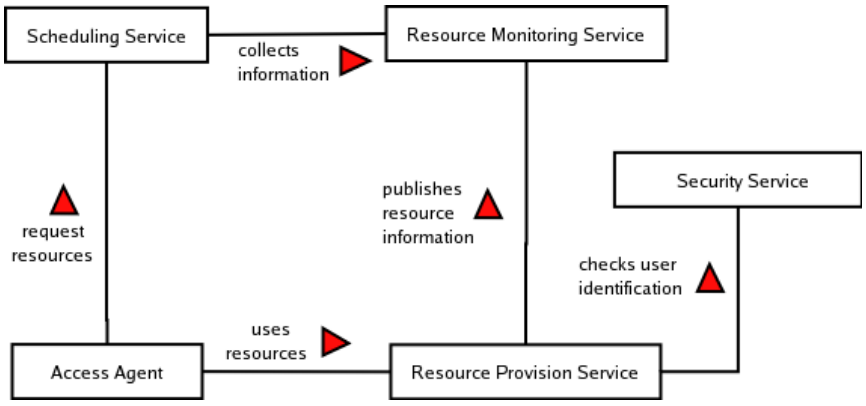

Figure 3: Relationships between the Grid modules.

and providing information about this state for other services. It monitors which resource providers are currently available and the state of their resources, such as processor usage and free memory. When requested, it can provide information about these resources, for example, when the scheduler needs to find a suitable node for the execution of an application. This service can also keep the resource usage and offering history for each user. This value can then be used by the scheduler to assign priorities to execution requests.

To keep the information up to date, this service receives periodic updates from resource providers concerning their resource availability status. This updates also serve for the monitoring service to detect when resource providers are not reachable. This can happen due to failures in the network or hardware and software failures in the node. The service then reports the errors for the scheduling service so that it can reschedule the processes that were running in these nodes for execution on another nodes.

- The *scheduling service* manages the available resources in the Grid. It is responsible for receiving application execution requests and, based on information provided by the resource monitoring service, selecting a node that satisfies the application requirements. When there are competing requests, the scheduler can define user priorities, which can be computed using factors such as the user class, resource usage history and resource sharing history.

- The *security service* is responsible for three major tasks: (1) protecting shared resources, so that a user who share his resources with the Grid does not suffer the effects of a rogue application, (2) managing users identities, so that application ownership can be established, enabling relations of trust and accountability, and (3) securing Grid communications, providing information confidentiality and integrity.

- Support for parallel applications is provided by libraries implementing parallel computing models, such as MPI [For93], BSP [Val90], and PVM [Sun90]. Those libraries are typically grid-specific versions implementing the same APIs defined by stand-alone versions of these

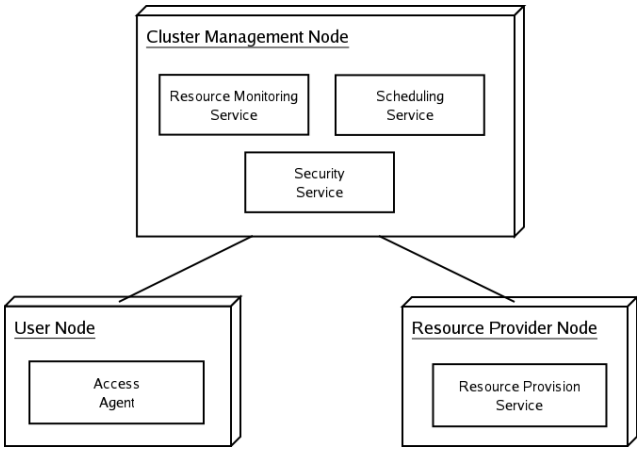

Figure 4: Typical Grid organization.

parallel libraries, thus allowing applications to run on the Grid with little or even no modifications.

CRC cards describing the Grid modules are presented in Figure 2. Figure 3 shows the interactions between Grid modules and Figure 4 shows a typical Grid organization.

Dynamics

Scenario I: this scenario, shown in Figure 5, illustrates the resource monitoring service receiving resource availability information from resource providers and responding to a query from the scheduling service.

- Resource providers send periodic updates with their resource offerings and usage.
- The scheduling service queries the resource monitoring service about resource availability.
- The resource monitoring service searches for available resources which satisfy the query. If resources are found, it returns their locations. Otherwise it returns a *resource not found* message.
- Resource providers keep sending periodic updates about its resource offerings and usage.

Scenario II: illustrates the Grid behavior when an access agent requests a single-node application execution (shown in Figure 6). In this example, we consider that no errors occur during application execution.

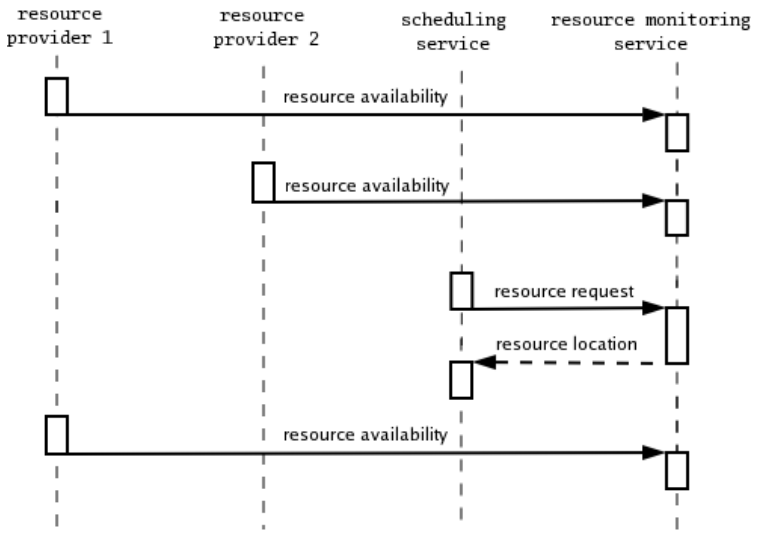

Figure 5: **Scenario I**: resource monitoring service.

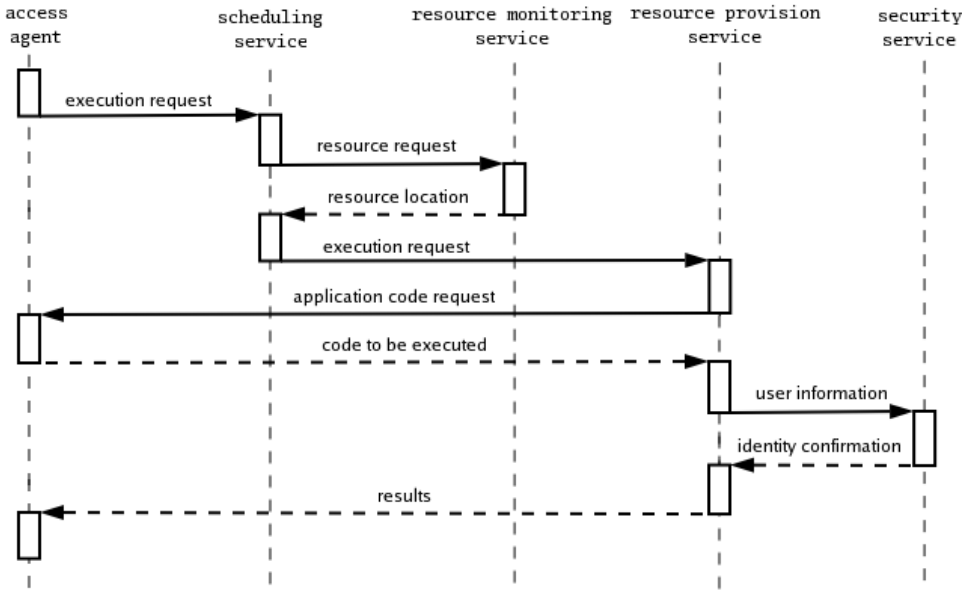

Figure 6: **Scenario II**: successful execution request.

- 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 a node meeting the resource 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 provider downloads the application code from the access agent that submitted the execution request or from an application repository.
- The resource provider checks the identity and permissions of the code owner with help of the security service.
- The resource provider executes the application. When the execution is finished, the results are sent back to the access agent.

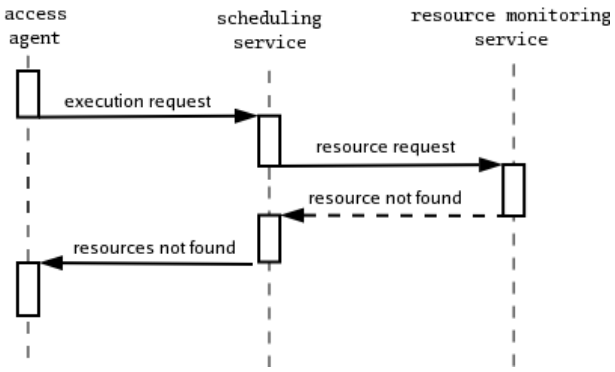

Figure 7: **Scenario III**: required resources not found.

Scenario III: illustrates the Grid behavior when the resources necessary for an application execution are not found (shown in Figure 7).

- The access agent sends an application execution request to the scheduling service, possibly with some information about the required resources.
- The scheduling service queries the resource monitoring service about resource availability.
- The resource monitoring service determines that it is not possible to satisfy the query with the available resources. It then returns a *resource not found* message.
- The scheduling service sends a failure report to the access agent. It is now the user's responsibility to decide what to do.

Implementation

Here we present some guidelines for implementing the Grid pattern.

- *Communication infrastructure:* the basic requirements of this infrastructure are that it work on all the platforms where Grid components will be installed and that it provide some form of reliable communication mechanism, such as remote procedures calls (RPC). Higher levels of abstraction in this layer eases the development of Grid components built on top of it.

  When implementing this infrastructure, it is easier to use an existing communication infrastructure, such as CORBA [Obj02a]. CORBA is a mature and robust object-oriented industry standard that is platform and language independent, and permits that communication among components be done through remote method invocations. Also, it provides various services that may be helpful for the implementation, such as naming, trading, security, and transaction services. Other existing communication infrastructures include DCOM, Java RMI, and .NET. The drawback of these infrastructures is that they are either platform or language dependent.

- *Access agent:* is implemented as a proxy that takes execution requests and forwards it to the Grid. It must provide a well defined API from where it is possible to make execution requests, specify application requirements, monitor application execution, and collect executions results.

  – *Submitting application code:* there are two possibilities: (1) the access agent directly uploads the application code to the resource providers selected for execution, or (2) the access agent first uploads the application code to an *application repository* used for application storage. The resource providers can then download the application code from this repository. The first option is the simpler, but the second gives more flexibility.

    Independent of the chosen method for application submission, it is also necessary to send an user certificate with the application. The resource provider who receives the application for execution can then check this certificate with a central authority provided by the security service.

  – *Format for application submission:* it is necessary to define the format of the applications to be submitted for execution. A simple possibility is submitting binary code ready for execution. The advantage is that it is ready to use, not requiring the compilation of the code on the resource providers. However, binary code is machine and operating system specific, and can require specific versions of shared libraries. If the Grid is composed of highly heterogeneous systems, a better approach is for the access agent to send the source code for compilation on the machines where the application will be executed. However, the need to compile the application in the target machine uses resources on that machine, increases system complexity, and also imposes that a compliant compiler be available in each of the target machines.

    A third possibility is to provide a virtual machine to interpret code pre-compiled in a platform independent format (e.g. Java or Smalltalk bytecode). To improve perfor-

mance, the virtual machine compiles this bytecode into native code just before application execution. The advantage is that application written in these languages can run in any platform where a virtual machine is available. But this advantage comes at a performance penalty and a higher level of resource consumption on the resource providers.

  - *User interface:* although not part of the access agent, it is important that a tool providing an interface for user interaction with the Grid be available. This interface can be implemented as a shell-like interface, a graphical interface or a Web portal.

- *Resource provision service:* must have a small memory footprint. Since it will be executed permanently in each of the resource providing nodes, it is important to assure that resource owners do not perceive any degradation in the quality of service provided by their machines. Consequently, if the implementation of this service uses external libraries or applications, these should be the smallest possible. Thus, this service should be implemented in programming languages that can lead to small memory footprint (such as C and C++) and do not require the execution of a large virtual machine.

  As was mentioned in the Structure section, this service is responsible for retrieving application code and starting its execution, controlling the local resource usage, and providing information about its resource offerings:

  - *Retrieving application code:* it can be implemented by either downloading the code directly from the access agent or from an application repository, as mentioned for the access agent. This code download is performed when the resource provider receives the execution request from the scheduler. After downloading this code, the last step before executing it is to check the identity and permissions of the application owner with the security service by sending the user certificate that was downloaded with the application code.
  - *Starting application execution:* the main issue is the format of the retrieved code. If it is the source-code of the application, the resource provider must compile and report compiler errors back to the scheduler. The scheduler would them report the error back to the access agent. If the application format is binary code ready for execution, the resource provision service only has to create a new process for executing the code.
  - *Providing resource usage information:* the resource providers must periodically send updates for the resource monitoring service regarding resource usage on the nodes, including the amount of resources used by jobs from different Grid users.

    An important decision is the time interval between updates. A shorter interval leads to higher network traffic, while longer intervals can lead to the information becoming stale. An adaptive approach can be used, dynamically choosing the best interval.
  - *Specifying constraints on resource sharing:* this service must provide a user interface that allows the resource owner to specify constraints on resource sharing. It should provide the possibility of specifying restrictions such as the users who can access the resources, the period of the day in which the resources can be shared, and the amount of resources that can be shared.

– *Managing of local resources:* a local scheduler manages the local resources. This local scheduler can be the same used by the operating system, with the processes submitted by the Grid being treated as background processes.

  In the case of shared workstations, there is the situation where the owner of a node decides to use the workstation to do his daily work. The workstation owner should not perceive a substantial drop in performance, so the executing process must be killed or suspended. The service must then send an error message to the scheduler, which will then either allow for the migration[2] of the process to another node or reschedule it for execution from the beginning.

– *Returning execution results:* a Grid process will typically put its results in one or more files, thus all this service has to do is to send these files back to the access agent. A special case occurs when the process fails during execution. In this case, the service must send an error report back to the scheduler which, depending on the error, will schedule the process for execution or send an error message for the access agent.

– *Usage patterns:* besides the periodical updates on resource availability, resource providers may contain more elaborate mechanisms for determining node usage patterns. For example, the mechanism can collect usage information on each node for a certain time interval and, through the use of clustering or time series algorithms, determine usage patterns that can help to predict the node usage for a future period of time. This data can be used by the scheduling service to improve resource usage efficiency. For example, if a usage pattern indicates that during the morning a machine usually remains idle for only short intervals, it would be unwise for the scheduling service to select this machine for the execution of a long running application.

- *Resource monitoring service:* needs to provide interfaces for both queries and updates regarding availability of the resources from resource providers. When queried, the resource monitoring service must perform a search for a set of nodes containing resources that satisfy the requirements specified in the query and return this set of nodes. In the case of updates from resource providers, it just updates its database. The resource usage and offering history for each user can be implemented by a simple resource usage index that is incremented when the user shares his private resources and decremented when the user uses Grid resources.

  In case of failure of Grid nodes, the resource monitoring service needs to inform the scheduler about the jobs that were running in that nodes. Information about the running jobs in each node is obtained from the scheduler when these jobs are scheduled for execution, and maintained by the resource monitoring service throughout their execution.

  For the storage of dynamic resource availability information, it is better to use existing solutions, such as the CORBA Trading Service [Obj00] or LDAP [YHK95]. Another possibility is to implement the service using a database, and provide some front-end for queries and updates.

[2] Process migration issues are discussed later in this section.

- *Scheduling service:* must employ a scheduling strategy that balances requests of users who need lots of computing power and users who seldom submit applications. This can be done by assigning user priorities that are modulated by the amount of resources consumed by each user. This modulation can be achieved using a resource modulation index maintained by the resource monitoring service. A simple strategy would be to prioritize jobs according to their owners' priority. Care must be taken to schedule simultaneously the jobs that compose a parallel application. By using the Strategy [GHJV95] pattern it should be easy to employ different scheduling strategies. The scheduling service must provide APIs for access agents to submit execution requests.

  The service should return an error to the user when the resources required to run the application cannot be found. The error message may provide some information about the available resources, so that the user can decide what to do. In the case an error occurs during the execution of a job, different approaches can be taken depending on the error. For a crash failure on the node where the job was executing, it is a good idea to reschedule the process for execution is another node. But in the case of an error in the application code such as a segmentation fault, a error message should be returned to the access client.

- *Security service:* resource providers can be protected by limiting system privileges for Grid applications. These restrictions can be implemented by intercepting system calls or using a virtual machine. An important design decision is how restrictive the environment should be. If the Grid makes use of user workstations, the security provided for resource owners must be a top priority. Typical restrictions include forking new processes, accessing peripherals such as printers, and obtaining free access to the filesystem. The resource provision service must provide an tool that allow the resource owner to specify these security restrictions.

  User authentication can be done through the use of certificates managed by a central authority. Resource providers can then verify user certificates with this central authority before executing Grid applications.

  The implementation of both user authentication and data encryption can be greatly simplified by reusing existing 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 a sophisticated alternative, but it can be used only if the communication infrastructure is also based on CORBA.

- *Process Migration:* since the Grid is composed of resources that can become unavailable at any moment, some mechanism to allow the progress of application execution in such a situation is necessary. That is, when migrating the application to a new node, it is important to resume its execution from an intermediate state, not from the beginning of its execution. In the case of parallel applications this is particularly important because, depending on the case, the failure of a single process can require that all the processes from the application be reinitialized. This occurs because message exchanges cause dependencies among the processes. For a parallel application containing many processes running on different nodes, the failure

rate would be too high.

A mechanism that allows this kind of migration is *Checkpointing* [EAWJ02]. It consists in periodically saving the application state as a checkpoint, allowing recovering application execution from the last saved checkpoint. The main issues are that this mechanism is difficult to implement and incurs an overhead in application execution time.

- The implementation of libraries for parallel programming can be done either from scratch or by modifying existing libraries for using of the Grid communication, resource monitoring, and scheduling infrastructures.

  For the implementation of process migration, it is necessary to consider the interprocess dependencies caused by message exchanges among application processes. A simple way to solve this problem is to synchronize all processes that exchanged messages since the last saved checkpoint.

- The deployment of the Grid pattern is as follows: the resource provision service is started on each machine that will export its resources to the Grid. Access agents are deployed on machines belonging to users that submit applications for execution. It is necessary to have a node where the scheduling, resource monitoring, and security services will run. This node has to be permanently available. If desired, this node availability constraint can be relaxed by adding redundancy to the resource monitoring, scheduling and security services, but this requires a more complex implementation of these services.

Variants

- *InterCluster Architectures:*

  The Grid pattern here presented, composed of centralized resource monitoring and scheduling services, scales well up to a few hundred nodes. When connecting more machines, this architecture shows some limitations. The main issue is that node management will be under a single administration. For a Grid composed of thousands of machines this can become a big problem. Another issue is that if the node containing the resource monitoring service crashes, all nodes in the Grid will become unreachable for a certain period of time.

  Consequently, if one needs to interconnect thousands of machines it is necessary to use a more scalable architecture. A possible solution is to divide the machines in clusters. Each cluster would then have its own resource monitoring and scheduling service, which need to be extended in order to communicate with other schedulers and resource monitoring services to provide inter-cluster integration. These clusters are organized in an inter-cluster architecture, for example, a static hierarchical or dynamic peer-to-peer architecture. The former is simpler to implement, while the latter is more flexible.

  Each cluster has its own scheduling, resource monitoring and security services. Intracluster schedulers try to execute requests from clients within its cluster. In case this is not possible,

it then contacts resource monitoring services from neighboring clusters to search for available resources in that clusters. If resources are found, the application execution request is forwarded to the scheduler of the cluster with available resources.

Known Uses

Globus [FK97] is a grid computing system that provides a collection of services necessary for developers to write applications to execute on Globus grids. It is built as a toolkit, which allows the incremental addition of functionalities for the development of grid applications. It implements most of the services described in this pattern. It has a resource monitoring service called **MDS** (Monitoring and Discovery Service) [CFFK01], a scheduling service called **GRAM** (Globus Resource Allocation Manager) [CFK+98], certificate-based authentication and support for some parallel programming APIs.

Condor [LLM88] allows the integration of many computational resources to create a cluster for executing applications. Condor emphasizes the use of shared resources that remain idle for a significant amount of time, such as workstations, to produce useful computations. The system was implemented using plain sockets for communication. Scheduling is performed by a central coordinator, which sends jobs to the machines offering its resources. These jobs are then put on a local execution queue. The conditions on which the resources can be used can be specified by the resource owner. Single process applications 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 implements most of the services of the Grid pattern. Its resource monitoring and scheduling services are implemented together as the *Global Resource Manager* (**GRM**). The resource provision service is called *Local Resource Manager* (**LRM**). Other key current InteGrade features are the use of CORBA for the communication infrastructure and a lightweight Resource Provision Service that imposes a small overhead on shared resources. Future releases will also include transparent support for checkpointing of parallel applications using the InteGrade APIs for parallel programming, support for usage pattern recognition and analysis, and a security service including user authentication and secure communication channels.

MyGrid [CPC+03] is a simple grid system which allows the execution of bag of tasks applications. The system is written in Java and uses RMI for communication. Its two main components are the scheduler and the *GridMachineInterface* (**GuM**). The scheduler has a functionality similar to the scheduling service from the Grid pattern while GuM implements the access agent and resource provision service.

Consequences

The Grid pattern presents some important advantages:

- *Reusability:* once implemented, the Grid infrastructure will work 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 hides the specific details on communications, computer architectures, and operating systems. Consequently, the applications can be developed much more easily, without the need to worry about details of communication and heterogeneous computer environments.

- *Easy application deployment:* the Grid manages the details of resource allocation, scheduling, and application deployment. This facilitates the execution process, since the user needs not to worry about the details of reserving computing resources and deploying the application on the nodes on which it will be executed.

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

- *Integration of dispersed resources:* the use of the Grid eases the integration of dispersed resources, including geographically distant nodes. In the cases these resources are spread across administrative domains, different policies for resource sharing, such as access restrictions, can be implemented for each domain.

Some disadvantages of using the pattern:

- *High complexity:* the implementation of the Grid pattern implies many problems that still do not have a consolidated solution. Although this problem is applicable for distributed systems in general, in the case of the Grid it needs to employ much more general and comprehensive solutions. Therefore, this task needs a considerable amount of time and effort of talented people, what can incur in a high cost. The investment on this system must be justified by the use on many applications and/or environments.

- *Necessity to change applications:* to take advantage of the Grid services, it may be necessary to alter parts of the application source. This requires access to the application source code, something that is rare on commercial applications. Even when the source code is available, its adaptation for the Grid 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.

- *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 distributed applications are limited.

Related Patterns

The **Broker** pattern [BMR+96] has similarities with the Grid pattern. Like the Grid, its goal is to encapsulate several details regarding the implementation of distributed systems, simplifying the development of applications. But, differently from the Grid, the Broker is recommended for simpler applications, such as business systems based on client/server architectures. Consequently, it does not require many of the characteristics available on the Grid, 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 pattern. The InteGrade system uses CORBA, an implementation of the Broker pattern, as the basis for its communication.

The **Master-Slave** pattern [BMR+96] also consists of having a central coordinator that distributes tasks for execution on servants and get the results back. But differently from the Grid pattern, the main focus of the Master-Slave pattern is multiprocessor architectures. It only applies to problems that can be solved by the 'divide and conquer' approach. But this specificity allows a simpler system that can be optimized for determined computer architectures, improving execution performance.

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. His innumerable suggestions were invaluable for improving the quality of this pattern.

References

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern - Oriented System Architecture: a System of Patterns*, chapter 2.3. 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.

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, and Jacques Sauvé. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of the ICCP'2003 - International Conference on Parallel Processing*, pages 407–, October 2003.

[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.

[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.

[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.