e-Informatica Software Engineering Journal, Volume 6, Issue 1, 2012, pages: 39–45, DOI 10.5277/e-Inf120103

# Middleware Architecture for the Interconnection of Distributed and Parallel Systems

Ovidiu Gherman*, Stefan Gheorghe Pentiuc*
**Electrical Engineering and Computer Sciences Faculty, "Stefan cel Mare" University*
ovidiug@usv.ro, pentiuc@eed.usv.ro

**Abstract**

Grid computing is a fast evolving technology, bringing more computing power to its users. Two main directions are observable: creating dedicated supercomputers for scientific and commercial tasks and creating distributed commodity-based systems. The first ones are usually much expensive, but have the advantage of performance, better control and uniformity in platforms. The second one is more affordable but lacks in flexibility and easy maintenance. The computing necessities that often require supplementary computing power for certain time periods are better satisfied by interconnecting available resources than buying new, expensive ones. But interconnecting platforms – sometimes radically different – can be a difficult task. The proliferation of hybrid parallel computing systems can be even more complicated because it puts in contact systems with various operating flows at the parallelism level. In this frame, the present article proposes a new middleware architecture that can connect multiple parallel or distributed resources, of different types, allowing unitary resource utilization and reservation for the user's jobs. The new architecture is described functionally and structurally.

## 1. Introduction

This article presents a middleware architecture that can connect a diversity of grid and parallel systems so that the users can run seamless jobs on multiple platforms, of different architectures, the compiling and executing part being sourced to the suitable resources assigned by a centralized (or decentralized) manager (broker). The main scope is interconnecting clusters of computers with different architectures and platforms (for example as MPI and hybrid Cell-based systems) so that the access will be transparent and uniform to the user, without the problems arising from the use of multiple computing clusters [1].

Certain acronyms will be used in the next pages: MPI – Message Passing Interface (communication protocol and specification set regarding communications between processes in parallel computers; popular software implementation are OpenMPI, MPICH1/2 and LAM-MPI), Cell and PowerXCell8i microprocessors manufactured by an alliance led by IBM (International Business Machines), hybrid systems (computing platforms that seamlessly integrates multiple CPU architectures), HPC – high performance computing, SSH/SFTP (Secure Shell and Secure File Transfer Protocol – widely used communication protocols in computers data transfers) SPF (single point of failure, showing a critical component that can disrupt or stop the entire system from normal work) and Beowulf systems (parallel computer systems built from inexpensive PCs).

Using a middleware (that offers a set of services) has the advantage of being easier to implement and can be installed on top of the already existent equipment (both hardware and software). Being message-oriented, can be easily extended and allows dynamic reconfiguration of the platform (for example when new resources are brought in the grid or crash and are removed from the available pool). The middleware can be extended so that new facilities can be attached. Once a new resource is added to the resource availability pool and properly set up (depending on the particular configuration of the software environment on that resource) it can be chosen to run certain jobs (in a generic way or a specific one – if it has a desired particularity).

## 2. Proposed Middleware Architecture

### 2.1. Introduction

The necessity of using large parallel and distributed computing systems required the creation of computing clusters – both homogeneous and heterogeneous regarding the distribution of hardware and software components. The most difficult step is to control and manage them efficiently and satisfactory for the user – goals that sometimes are opposite.

The proposed architecture wants to be a “glue” between parallel computing platforms that use the grid infrastructure already on the local resources (such as parallel clusters or Beowulf systems using MPI platforms – OpenMPI, MPICH1/2, LAM-MPI – and/or hybrid platforms like MPI on global level and PowerXCell locally [1]) and to extend the functionality of these systems, in the same time connecting them in an unitary fashion, transparently to the user. This middleware is built on top of the local operating systems in order to benefit from the security and optimization layout of the OS and parallel middleware, allowing fast development and optimization.

The middleware architecture provides an easy access method to numerous resources with different specifications, in which a set of services are made available to the users, services that allow to define a set of attributes required for the execution of the programs (and the programs themselves), with the actual process of allocation and reservation of certain resources being made automatically. Remote compilation and execution allows writing only one source code for a given project (respecting the characteristics of the desired target machine) and thus creating platform-agnostic programs.

### 2.2. Resource Classification and Performance Criteria

The proposed architecture uses a set of different resources that are allocated to the incoming jobs. For a more suitable planning, every resource can be scored [2,3] as to ascertain the trustfulness of the given resource regarding the online time ratio $V$ and the success ratio $G$ (and to quantify the level of QoS compliance). This way, the resources can be classified for better QoS compliance (the most reliable resources are the most used). Although there are more parameters that can be used to measure quantitative the QoS level [4], the most meaningful in this case are:

$$V = \frac{\text{time}_{\text{online}}}{\text{time}_{\text{total}}} \tag{1}$$

and:

$$G = \frac{\text{jobs}_{\text{succesful}}}{\text{jobs}_{\text{total}}} \tag{2}$$

### 2.3. Layered Architecture

The proposed middleware amalgamates multiple clusters and distributed general-use systems into a unitary platform, under a centralized or decentralized management system.

The parallel clusters have – usually – a local management system that is highly optimized [5]. When multiple such clusters are interconnected (even if those clusters are homogenous internally, having identical components in the nodes and a global parallel environment), the differences in the platform, operating systems, middleware for parallel applications or administration policies can generate difficulties in the competent and automatic allocation of the available resources and in the seamless running of the client’s applications. This allows a better level of quality of service also by simplifying the overall architecture and hiding the QoS penalties induced by the computing systems themselves. Their optimization is executed apart, by the rightful administrators, the management system (including the resource broker) monitoring and scoring every resource accordingly. This makes easier to deploy the middleware across multiple parallel and distributed systems and to create a unique grid infrastructure. The scoring allows to employ specific selection algorithms that will reserve the most appropriate resource (depending on the user's requirements and his job's requirements), making possible to guarantee a certain level of QoS. Supplementary modules can be employed (future work) to obtain a greater redundancy in executing jobs (checkpoint, job migration, etc.) [6] and a superior QoS level, even for volatile grids.

The proposed architecture uses multiple layers to describe the level of operation in the system, as in Fig. 1.

The middleware uses the operating systems mechanism to operate with the user files and the communication systems, compiling and running them in the parallel application environments installed locally on every cluster (resource) (usually MPI or MPI/hybrid) [7, 8]. It also employs the security layer already in use at the cluster/node level. The communication protocols are SSH/SCP/SFTP, used by virtually every *nix OS used on the HPC (High Performance Computing) systems and distributed systems in use. Thus, no other security holes will be opened in the security infrastructure, the systems will be properly patched as part of the maintenance requirements and the deployment will benefit from a stable and tested protocol/security module. The middleware will use the well known and reliable facilities offered by the system itself.

Similar application are usually deployed for dedicated platforms – high throughput computing like Condor Project [9] or distribution for Beowulf clusters (such as Rocks Cluster). These solutions are dedicated to job scheduling or cluster management, lacking a lightweight approach that allows using volatile resources or diverse communities of parallel and distributed systems.

The deployment of the middleware will be on top of the local grid infrastructure, at three levels: user, resource manager (broker) and resource. It will use every resource in conformity with its particularities.

## 2.4. Architecture, Modules and Functioning

The architecture of the middleware has three main areas.

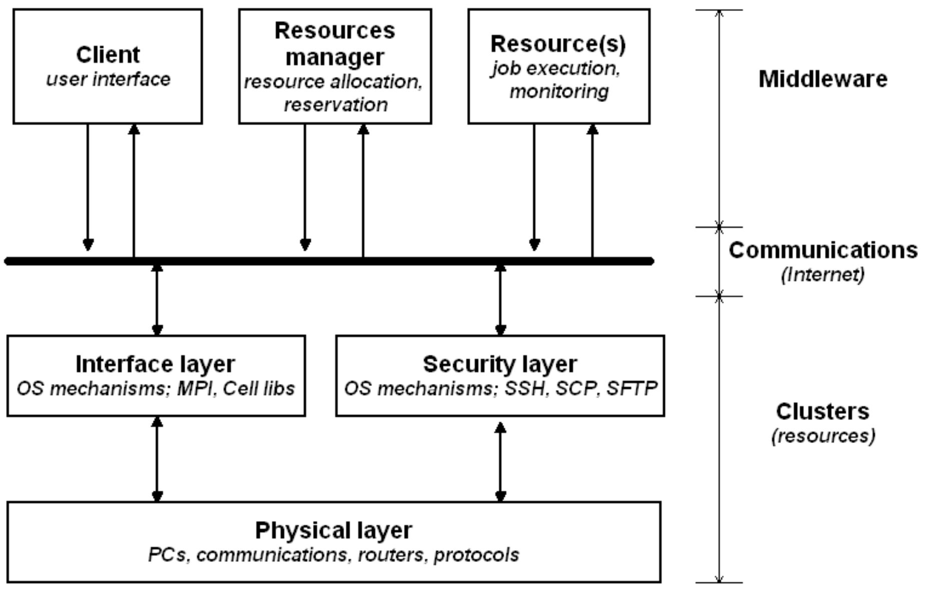
Client
user interface
Resources manager
resource allocation, reservation
Resource(s)
job execution, monitoring
Middleware
Communications
(Internet)
Interface layer
OS mechanisms; MPI, Cell libs
Security layer
OS mechanisms; SSH, SCP, SFTP
Clusters
(resources)
Physical layer
PCs, communications, routers, protocols

Figure 1. The layered architecture of the proposed middleware, built on top of the communication and security mechanisms offered by the underlying operating systems.

*The user (client) area*, which represents the only method of interaction between an authorized user and the resources (without implicating a system administrator). Once the middleware is set and configured, the interface allows submitting jobs (along with the required attributes) and retrieving the result of the jobs either positive or negative if the job fails from user fault (wrong instructions or conditions, errors in programming, errors in algorithms, etc.) or system fault (node malfunction, environment problems, software failure, etc.) – all transparent to the user. The error messages produced by the operating system/middleware/software interfaces can be automatically analyzed or logged for further study (since the module operates on top of the software platform, it can record the activities associated with the client applications). If no suitable resource is found the job is aborted. The user must specify – along the source code of the program – a set of attributes that will define the requirements of the job (number of CPUs, special software requirements, and particular hardware architectures – such as the hybrid ones). These requirements, along with the user's profile (access rights, history, program's nature and execution length) will be factored in selecting a suitable resource (with immediate execution or advance reservation).

*The resource manager (the resource broker) area* must monitor the available resources (especially the volatiles ones – resources that often come online/offline, completely or partially, or those that have heavily variable performance) in order to make an accurate and valid selection (suitable to the user and efficient for the platform). Also, it must employ an algorithm to select the suitable resources (based on resource specifications, job requirements and user profile). Because of the modularity of the middleware, such multiple algorithms can be used (from the simple to the most complex ones), allowing even to benchmark those algorithms.

The resource manager must have a database with the resources' behaviour in order to make accurate prediction regarding the suitability of the resources. The monitoring module must be permanently online and in communication with the resources, activity that can lead to signification overhead at the manager level (in computation and communication alike). It is best to use a dedicated machine for this module. Since the middlware uses communication methods already present at the system-level, the informations can be send as commands or data strings.

Also, a QoS module must be used to ensure that the selected resource will run the job in time. If a better performance than best-effort is sought, a series of estimates must be generated for every job – the estimated computation length of the job and a programmed date for the start of the program (for advanced reservation). The job estimates can be generated by the user (and mediated by that user's history in giving accurate predictions) for example. If a job is not executed in the allotted time, the event is logged and used for scoring the resource/algorithm.

*The resource area* will receive user jobs (with the attributes and the broker's instructions) and will compile and run the requested jobs, in conformity with the local parallel environment. Because the compilation is local, the user has the flexibility in writing the source code (providing that he specifies correctly the attributes). Every resource (cluster or grid) will have its own environment, with specific commands, that will be given in the setup stage by the manager of that resource.

The resource area must also monitor the node behaviour and must report to the resource manager.

The proposed architecture of the middleware is described in Fig. 2.

The platform must be modular, flexible, in order to permit using multiple selection and planning algorithms to test their performance and the resource consumption on the given QoS specifications. This is more important for the volatile systems. If a parallel cluster is more reliable (it is homogenous and has a compact nature), the network outage (Internet) can influence greatly (and easily) the volatility of the system based on multiple parameters. There are 4 steps to be taken in order to work: user-resource manager communication, resource allocation, sending the job in accordance with the allocation scheme,

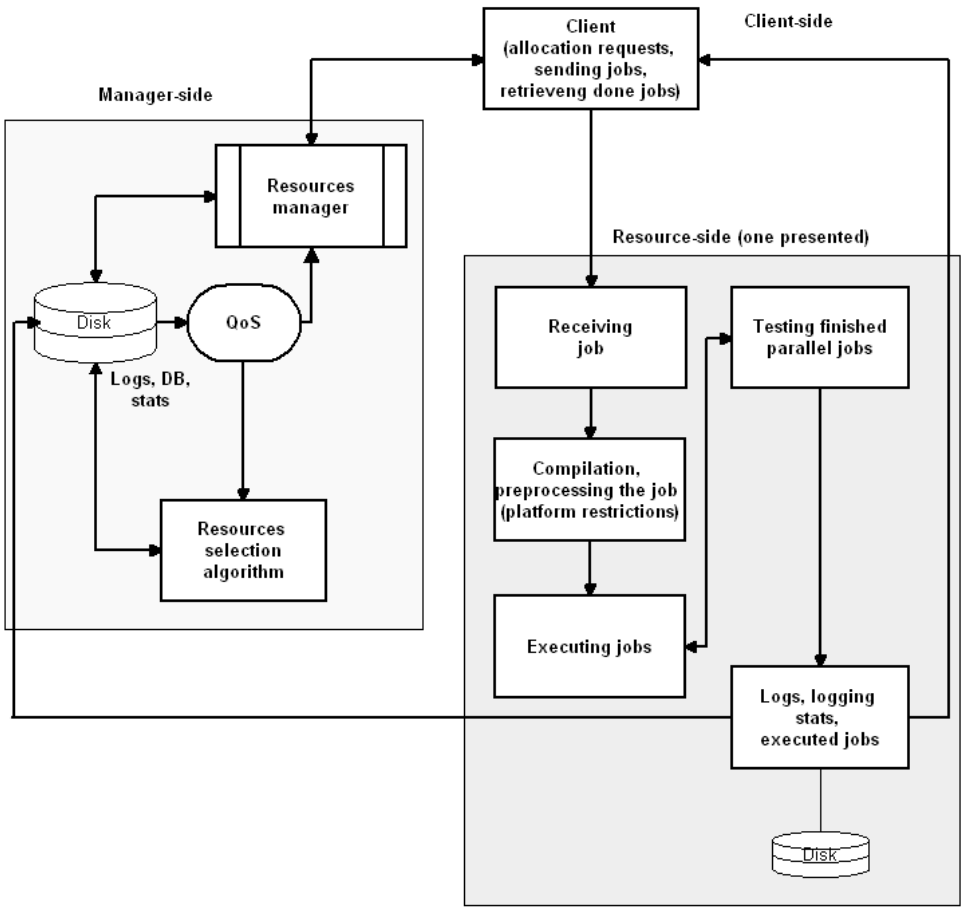
Client (allocation requests, sending jobs, retrieveng done jobs)
Client-side
Manager-side
Resources manager
Disk
QoS
Logs, DB, stats
Resources selection algorithm
Resource-side (one presented)
Receiving job
Testing finished parallel jobs
Compilation, preprocessing the job (platform restrictions)
Executing jobs
Logs, logging stats, executed jobs
Disk

Figure 2. The architecture of the proposed middleware (GLUE).

taking back the processed job. Each activity is logged and analyzed afterward.

One of the main vulnerability of this architecture is the use of a centralized unit to run the resource manager components. This inserts a single point of failure (SPF) vulnerability – in the event that the unit crashes or loses the Internet connection, the middleware becomes broken (requiring recovery from the previous states, possibly with loss of information). There are at least 2 possible methods to avoid this vulnerability:

– Using multiple resource managers that run synchronized; the loss of one unit is covered by one of the neighbours. Every job analysis and resource monitoring is repeated in all units (but only one is active), so that no information will be lost if the unit fails (Fig 3). To avoid the waste in terms of bandwidth and CPU time (mainly as redundancy) if a replication service for multiple manager modules is employed, only one manager will be active, logging the results of different actions; in the event of a crash, every other instance (selected by a default order) can take its place, losing only the current action (if any). This way, a certain degree of redundancy is assured and the possibility of a SPF is avoided.
– The resource manager can be completely decentralized. Every instance is running a certain number of connections (the users select the managers in conformity with a default order or randomly) so that the grid is decongested (using different units and different communication routes). For efficient planning, at certain time moments the databases must be synchronized (for efficiency and consistency) between the managers. If a unit crashes, its jobs and data can fallback to a backup unit (losing only the current operation, if any) or can be taken by a neighbour

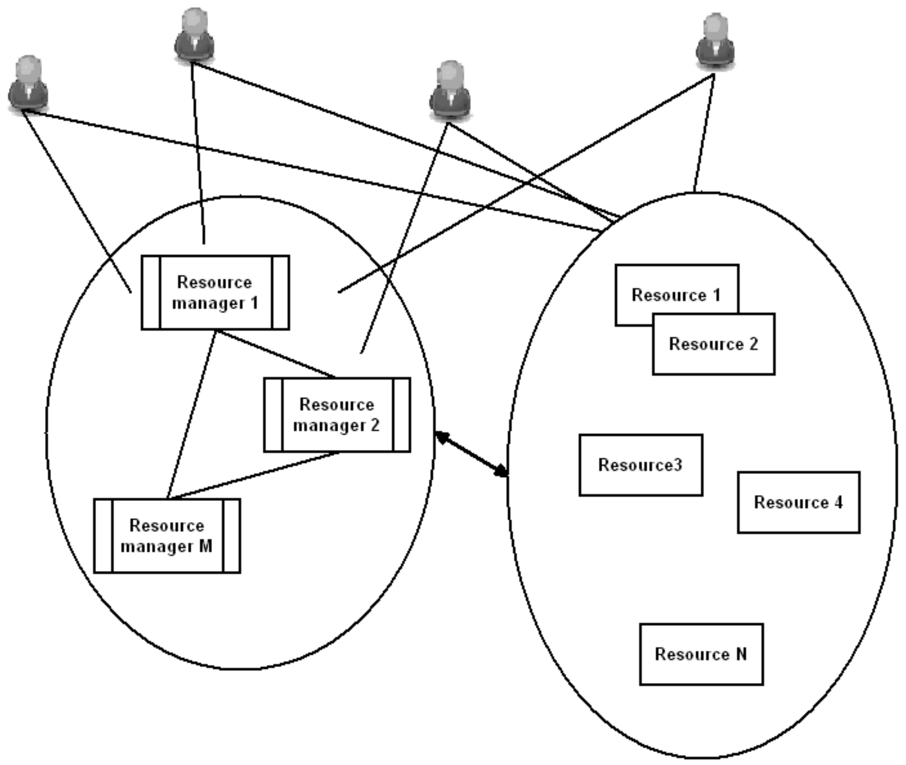

Figure 3. Schematic diagram using distributed (decentralized) resource managers.

unit. The users can download actualized lists for the updated resource managers.

## 3. Conclusions

The proposed architecture brings advantages for interconnecting clusters systems: the architecture is modular (with modifiable modules and the possibility to insert new selection algorithms), flexible and portable, being written using Perl scripting language (scripting language is flexible, portable and powerful, allowing access to the underlying architecture below the middleware level). Also, the architecture provides increased security because it uses a reliable security mechanism that is already in place at the OS level and allows management of different clusters and systems and easy upgrade to the software. The middleware allows transparent, seamless application build and execution for the user. Users can specify certain particularities of the intended target machine to help selection (for example when having commodity hardware based clusters or parallel and hybrid clusters). The application itself is written in Perl, a known scripting language that offers flexibility to the platform and allows designing the work modules (and the addition of new ones).

Some observed disadvantages are: the possibility of SPFs at the resource management level avoidable if the system is decentralized, as been observed [10]; the resources crashing can – missing a recovery solution – lose jobs (worsening the QoS compliance of the system); it cannot allow execution of big jobs across multiple resources (but allows the simultaneous execution of multiple jobs on a resource, providing there are enough free CPUs); computing the planning for resource allocation can generate an overhead for complex algorithms and an increased number of users and resources (although the algorithm can be parallelised for increased performance). Finally, the middleware has a rather specific purpose and reduced applicability outside it (e.g. not fit for commercial domain).

## 4. Future Work

The middleware will be expanded (functionality-wise) in the future, including new selection algorithms (for the resource manager), preparing the source code to be error-tolerant and stable on the diversity of available grid platforms, improving resource monitoring (increasing number of relevant logged parameters), possibility to state advanced reservation for certain jobs and plans and others and – possibly – methods of increasing the quality of service offered by preventing the loss of jobs (for example by using job replication).

## Acknowledgment

This paper was supported by the project "Doctoral Burses at USV", contract POSDRU/6/1.5/S/22, funded from European Social Fund for Operational Sectoral for Human Resources Development 2007-2013.

### References

[1] O. Gherman, I. Ungurean, and S. G. Pentiuc, "Principles of interconnecting a hybrid cluster in a grid system," *7th edition of National Scientific Conference Distributed Systems*, No. 7, Dec 2009.

[2] N. Fujimoto and K. Hagihara, "A comparison among grid scheduling algorithms for independent coarse-grained tasks," *International Symposium on Applications and Internet Workshps SAINT 2004*, 2004.

[3] I. Goiri, F. Julia, O. Fito, M. Macias, and J. Guitart, "Resource-level QoS metric for CPU-based guarantees in cloud providers," *7th International Workshop on Economics of Grids, Clouds, Systems and Services GECON 2010*, No. 7, 2010.

[4] S. Kalepu, S. Krishnaswamy, and S. W. Loke, "Verity: a QoS metric for selecting web services and providers," *4th International Conference on Web Information Systems Engineering Workshops WISEW'03*, No. 4, 2003.

[5] O. Gherman, I. Ungurean, S. G. Pentiuc, and O. Vultur, "Data communication in a HPC hybrid cluster and performance evaluation," *10th International Conference on Development and Application Systems DAS 2010*, No. 10, May 2010.

[6] C. Dabrowski, "Reliability in grid computing systems," *The Special Issue of the Open Grid Forum (OGF) Journal – Concurrency and Computation: Practice and Experience*, Vol. Volume 21, No. 8, 2009.

[7] K. Koch, "Roadrunner platform overview," Roadrunner Technical Seminar Series, Los Alamos National Laboratory, SUA, Technical report, 2008.

[8] C. Kessler, "Programming techniques for the cell processor," Multicore Day Seminar, Sweden, Technical report, 2009.

[9] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," http://www.cs.wisc.edu/condor/doc/beowulf-chapter-rev1.pdf, 2001, chapter 15, research project at the Unversity of Wisconsin-Madison.

[10] R. Buyya, D. Abramson, and J. Giddy, "An economy driven resource management architecture for global computational power grids," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA 2000*, 2000.