Design and Implementation of a Grid-Enabled Component Container for *CORBA Lightweight Components*⋆

Diego Sevilla[1], José M. García[1], Antonio Gómez[2]

[1] Department of Computer Engineering
[2] Department of Information and Communications Engineering
University of Murcia, Spain
{*dsevilla, jmgarcia*}*@ditec.um.es, skarmeta@dif.um.es*

Abstract Although Grid technology appears as a promising infrastructure for global computation and effective resource sharing. However, the development of Grid Applications is still based in traditional programming models such as MPI, making it difficult to provide a good level of software reuse and productivity. Moreover, the Grid offers an environment where the component technology can be applied to a greater extent than ever, due to the intrinsic security enforced by the Grid, allowing the creation of a successful component market. Component technology accelerates software development enforcing software reuse and sharing. In this article we present CORBA–$\mathcal{LC}$ and the design of its Container, that manage components and provides them with non-functional aspects such as security, concurrence, distribution, load balancing, fault tolerance, replication, data-parallelism, etc. Component implementors can thus focus only on the component functionality itself, independently of these aspects, provided by the Container. Moreover, we identify these non-functional aspects in the Grid Computing domain and show the current status of the implementation.

1 Introduction and Related Work

Grid technology [4] has emerged as a new paradigm for reusing the computing power available in organizations worldwide. Particularly, Grid toolkits like Globus [7] and frameworks like *Open Grid Services Architecture* (OGSA) [11] help establishing a standard framework for integrating new developments, services, users, organizations, and resources.

Within these frameworks, which offer the foundation for the development of the Grid, distributed component models fit seamlessly to provide a higher level services for integrating and reusing components and applications.

Component models allow developing parts of applications as independent components. These components can be connected together to build applications, and represent the unit of development, installation, deployment and reuse [16].

⋆ Partially supported by Spanish SENECA Foundation, Grant PB/13/FS/99.

Taken together, the benefits of both the Grid and components raise the level of reuse and resource availability, allowing the development of a "component market", in which all the organization offer their components and services.

Traditional component models such as *Enterprise Java Beans* (EJB) and the *CORBA Component Model* (CCM) [10] are not suited for Grid computing because the enterprise services overhead. Thus, other component models oriented towards the Grid have appeared, such as the *Common Component Architecture* (CCA) [1], the work of Rana et al. [8] and Furmento et al. [5]. However, these works do not offer a complete component model, neither packaging nor deployment models, making it difficult to manage applications and services in the Grid environment.

In this article we present the *CORBA Lightweight Components* (CORBA–$\mathcal{LC}$) distributed component model and study the design and implementation strategies for its component container.

2 The CORBA–$\mathcal{LC}$ Component Model

CORBA Lightweight Components (CORBA–$\mathcal{LC}$) [14,15] is a lightweight component model based on CORBA, sharing many features with the CORBA Component Model (CCM)[10].

The following are the main conceptual blocks of CORBA–$\mathcal{LC}$:

- **Components**. Components are the most important abstraction in CORBA–$\mathcal{LC}$. They are both a *binary package* that can be installed and managed by the system and a *component type*, which defines the characteristics of component instances (interfaces offered and needed, events, etc.) Component characteristics are exposed by the **Reflection Architecture**.
- **Containers and Component Framework**. Component instances are run within a run-time environment called **container**. Containers become the instances view of the world. Instances ask the container for the required services and it in turn informs the instance of its environment (its *context*). Component/container dialog is based on agreed local interfaces, thus conforming a component framework. The design and implementation strategies for the CORBA–$\mathcal{LC}$ containers are described in Section 3.
- **Packaging model**. The packaging allows to build self-contained binary units which can be installed and used independently. Components are packaged in ".ZIP" files containing the component itself and its description as IDL and XML files. The packaging allows storing different binaries of the same component to match different Hardware/Operating System/ORB.
- **Deployment and network model**. The deployment model describes the rules a set of components must follow to be installed and run in a set of network-interconnected machines in order to cooperate to perform a task. CORBA–$\mathcal{LC}$ deployment model is supported by a set of main concepts: **nodes**, the **reflection architecture**, the **network model**, the **distributed registry** and **applications**.

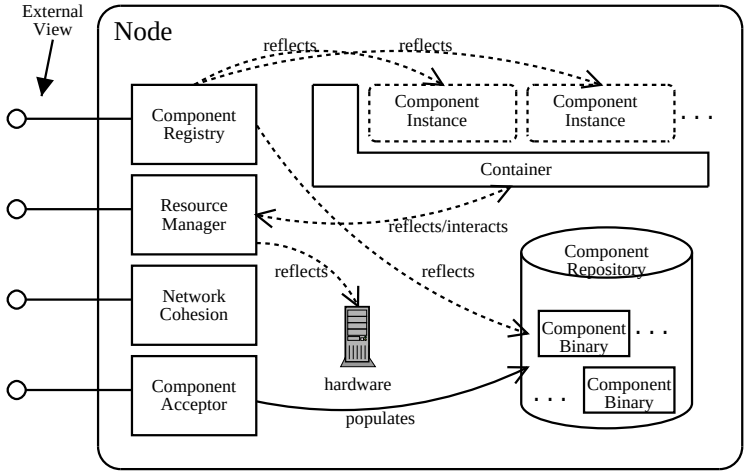

Figure 1. Logical Node Structure.

- **Nodes**. The CORBA–$\mathcal{LC}$ network model can be seen as a set of nodes (hosts) that collaborate in computations. Nodes maintain the logical network connection, encapsulate physical host information and constitute the external view of the internal properties of the host they are running on. Concretely, they offer (Fig. 1):
  - A way of obtaining both node static characteristics (such as CPU, Operating System type, ORB) and dynamic system information (such as CPU and memory load, available resources, etc.): ***Resource Manager*** interface.
  - A way of obtaining the external view of the local services: the ***Component Registry*** interface reflects the internal ***Component Repository*** and allows performing distributed component queries.
  - Hooks for accepting new components at run-time for local installation, instantiation and running [9] (***Component Acceptor*** interface).
  - Operations supporting the protocol for logical ***Network Cohesion***.
- **The Reflection Architecture**. Is composed of the meta-data given by the different node services:
  - The ***Component Registry*** provides information about (a) running components, (b) the set of component instances running in the node and the properties of each, and (c) how those instances are connected via ports (assemblies)[13]. This information is used when components, applications or visual builder tools need to obtain çinformation about components.
  - the ***Resource Manager*** in the node collaborates with the ***Container*** implementing initial placement of instances, migration/load balancing at run-time. Resource Manager also reflects the hardware static characteristics and dynamic resource usage and availability.

- **Network Model and The Distributed Registry**. The CORBA–$\mathcal{LC}$ deployment model is a network-centered model: The complete network is considered as a repository for resolving component requirements. Each host (node) in the system maintain a set of installed components in its ***Component Repository***, which become available to the whole network. When component instances require other components, the network can decide either to fetch the component to be locally installed, instantiated and run or to use it remotely. This network behavior is implemented by the ***Distributed Registry***. It stores information covering the resources available in the network as a whole.
- **Applications and Assembly**. In CORBA–$\mathcal{LC}$, ***applications*** are just special components. They are special because (1) they encapsulate the explicit rules to connect together certain components and their instances (*assembly*), and (2) they are created by users with the help of visual building tools. Thus, they can be considered as ***bootstrap*** components.

3 A Grid-Enabled Container for CORBA–$\mathcal{LC}$

Containers in CORBA–$\mathcal{LC}$ mediate between component instances and the infrastructure (both CORBA–$\mathcal{LC}$ runtime and the Grid middleware). Instances ask the container for needed resources (for instance, other components), and it, in turn, provides them with their *context*. Figure 2 shows the building blocks of a container, as well as its responsibilities within a call by the component client. Concretely, container responsibilities include [17]:

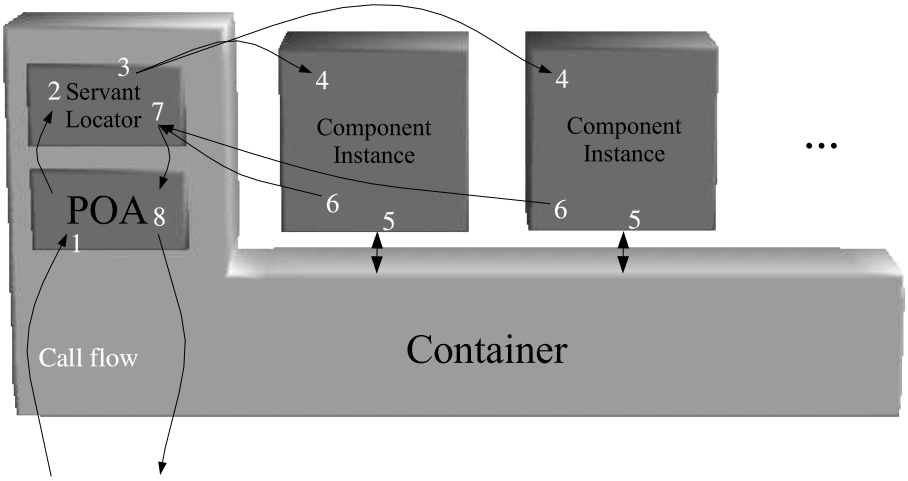

Figure 2. Component instances and interception within a Container.

- **Manage component instances**. It interacts with the component factory to maintain the set of active component instances, as well as instance activation and deactivation.
- **Provide a controlled execution environment for instances**. They obtain all the network resources (their *view* or *context*) from the container, becoming their representative into the network. The communication between container and instances is made through agreed local interfaces.
- **Provide transparent fault tolerance, security, migration, and load balancing**. The container intercepts all the calls made to the component instances it manages. This gives the container the chance to redirect the calls to the correct component or to group component instances to implement fault tolerance. There are two main strategies to implement this behavior:
  - **Interception**. Using CORBA as a foundation, *Portable Interceptors* [10] can be used to intercept every call to the instance. This is more flexible and generic, but inefficient.
  - **Code generation**. With this approach, an utility can take interface definitions and non-functional aspects of the component (described as XML files) and generate a customized container. The container becomes a wrapper for the component. This approach is more efficient because the container is made *for* the component. Moreover, the code generation can also convert the generated container into an *adapter* [6] to offer the component interfaces as Grid Services compliant with the OGSA specification [11].

Figure 2 shows the flow that a client call follows within the Container. The client call in intercepted by the Container's POA (1). The POA invokes the Servant Locator of the Container (2). This is in charge of locating the actual component instance for the call. If the instance is not activated, it will activate it. Alternatively, it may redirect the call to another host depending on the requirements on the application (replication, etc.) (3). The Component Instance now takes the control, executing the required operation (4). While its operation, the instance may call the container to obtain services from the network (5), such as other components or resources. The instance then returns the results of the operation (6), and the Servant Locator takes the control again (7). At this point, it can decide if to passivate the instance to save system resources. The POA finally returns the results to the caller. Points (3) to (7) represent interception points.

Containers follow the Aspect-Oriented Programming (AOP) philosophy [3]. Components specify the non-functional requirements (*aspects*) they need from the environment. This specification is made through XML files describing the characteristics of the component in terms of the defined aspects. Figure 3 shows the CORBA–$\mathcal{LC}$ tool chain. The user provides both the IDL definitions for the component and the XML file describing the component characteristics. This separation allows using traditional ORBs and IDL2 compilers instead of forcing to use a CORBA 3 implementation as CCM does. The CORBA–$\mathcal{LC}$ Code Generator generates the code that interacts with the container for doing aspect weaving, and the IDL Compiler generates traditional CORBA stubs and skeletons.

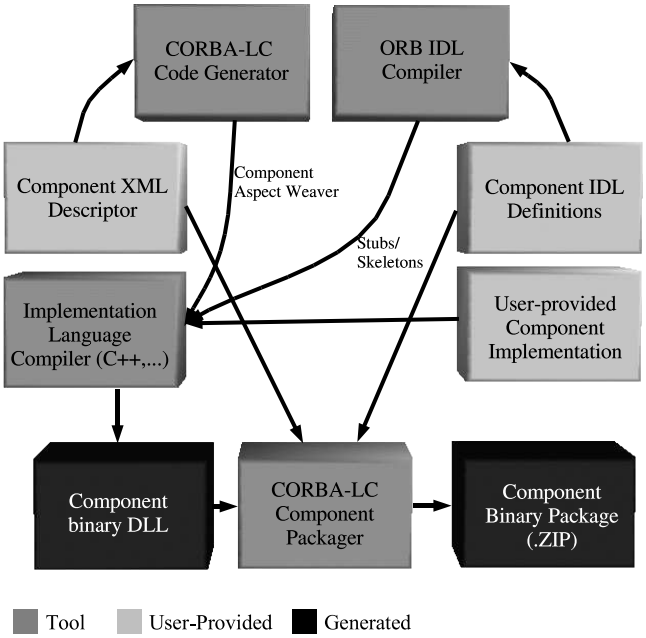

Figure 3. CORBA–$\mathcal{LC}$ tool chain.

These, together with the Component Implementation provided by the user, are compiled into a Binary DLL (Dynamic Link Library). Finally, the CORBA–$\mathcal{LC}$ Component Packager gets the DLL and the metadata of the component (XML and IDL) and packages it for distribution.

Figure 4 shows the role of the generated weaving code. The component instance is composed of the generated code and the code supplied by the user. The weaving code interacts with the container to provide the required aspects. For instance, if the user (or the application) decides to log the calls to this instance, the generated code will do the logging before calling the actual instance implementation provided by the user (The same can be applied for Security, distribution, etc.) Just after applying desired aspects, it calls the user implementation. The implementation uses the container to obtain resources from the system, such as other components and so on.

While traditional component models as EJB and CCM cover the needs of enterprise-oriented applications, we believe they are not suited for dealing with Grid or HPC applications, because of the burden of enterprise-oriented services such as persistence or transactions. Moreover, they have a fixed set of aspects, making them difficult to adapt to those environments.

Following the AOP paradigm, we have identified a set of aspects that components can specify so that the container can manage them effectively in a Grid environment. This list is not exhaustive, and is a result of our ongoing research on this area:

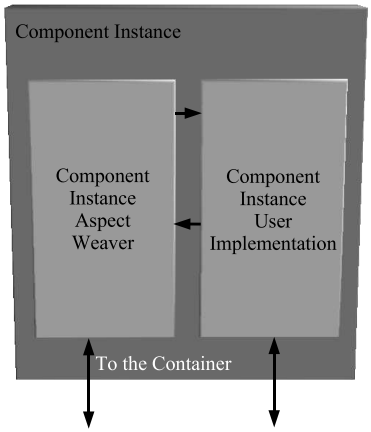

Figure 4. Internal view of a component instance.

- **Instance lifetime and service**. Defines the lifetime of instances. Helps the container to manage instances.
- **Integration with grid security**. Specifies the security restriction of this component. The container must ensure component security restrictions by leveraging the grid infrastructure.
- **Mobility**. If the component can travel or must be run remotely. The former allows physical component distribution. The latter is semantically equivalent to a OGSA service.
- **Fault Tolerance** and **Replication**. The container must ensure the level of fault tolerance required by the component (number of replicas, etc.)
- **Data Aggregation and distribution**. This is interesting for data-parallel components, which can specify how many *workers* they need for the realization of their work, and know how to join partial results. The container is in charge of finding the workers, delivering the data and bringing back the results.

4 Status and future work

Current status of CORBA–$\mathcal{LC}$ allows building components and connect them. We are currently researching in the area of aspects suited for grid computing and the design and implementation of the CORBA–$\mathcal{LC}$ container. Concretely, we are working in the following fields:

- Identification of aspects suitable for Grid computing, and its application to CORBA–$\mathcal{LC}$ and the Container and Code Generator. [2].
- The implementation of the Container and the different aspects such as Grid Security, leveraging the different Commodity Grid Kits (CoG) such as the CORBA CoG [12].

– We are implementing the CORBA–$\mathcal{LC}$ distributed deployment, which interacts with each component container to offer distributed services such as replication, load balancing and fault tolreance.

References

1. CCA Forum. *The Common Component Architecture Technical Specification - Version 1.0.* http://z.ca.sandia.gov/~cca-forum/gport-spec/.
2. F. Duclos, J. Estublier, and P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. In *International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.
3. J. Fabry. Distribution as a set of Cooperating Aspects. In *ECOOP'2000 Workshop on Distributed Objects Programming Paradigms*, June 2000.
4. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1999.
5. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Optimisation of Component-based Applications within a Grid Environment. In *SuperComputing 2001, Denver*, November 2001.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
7. *The Globus Project Home Page.* http://www.globus.org/.
8. M. Li, O. F. Rana, M. S. Shields, and D. W. Walker. A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components. In *Supercomputing'2000 Conference*, Dallas, TX, November 2000.
9. R. Marvie, P. Merle, and J-M. Geib. A Dynamic Platform for CORBA Component Based Applications. In *First Intl. Conf. on Software Engineering Applied to Networking and Parallel/Distributed Computing (SNPD'00)*, France, May 2000.
10. Object Management Group. *CORBA: Common Object Request Broker Architecture Specification, revision 3.0.1*, 2002. OMG Document formal/02-11-01.
11. *Open Grid Services Architecture.* http://www.globus.org/ogsa.
12. M. Parashar, G. von Laszewski, S. Verma, J. Gawor, K. Keahey, and N. Rehn. A CORBA Comodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 2002.
13. N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair. Towards a Reflective Component-based Middleware Architecture. In *ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, 2000.
14. D. Sevilla, J. M. García, and A. Gómez. CORBA Lightweight Components: A Model for Distributed Component-Based Heterogeneous Computation. In *EUROPAR'2001*, Manchester, UK, August 2001.
15. D. Sevilla, J. M. García, and A. Gómez. Design and Implementation Requirements for CORBA Lightweight Components. In *Metacomputing Systems and Applications Workshop (MSA'01)*, Valencia, Spain, September 2001.
16. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press, 1998.
17. M. Vadet and P. Merle. Les conteneurs ouverts dans les plates-formes à composants. In *Journées Composants, Besançon, France*, October 2001.