View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Juelich Shared Electronic Resources

John von Neumann Institute for Computing

Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA-LC Component Model

Diego Sevilla, José M. García, Antonio Gómez

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 347-354, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38

Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA–$\mathcal{LC}$ Component Model

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

[1] Department of Computer Engineering

[2] Department of Information and Communications Engineering
University of Murcia, Spain
E-mail: {dsevilla, jmgarcia}@ditec.um.es, skarmeta@dif.um.es

Programming abstractions, libraries and frameworks are needed to better approach the design and implementation of distributed High Performance Computing (HPC) applications, as the scale and number of distributed resources is growing. Moreover, when Quality of Service (QoS) requirements such as load balancing, efficient resource usage and fault tolerance have to be met, the resulting code is harder to develop, maintain, and reuse, as the code for providing the QoS requirements gets normally mixed with the functionality code. Component Technology, on the other hand, allows a better modularity and reusability of applications and even a better support for the development of distributed applications, as those applications can be partitioned in terms of components installed and running (deployed) in the different hosts participating in the system. Components also have requirements in forms of the aforementioned non-functional aspects. In our approach, the code for ensuring these aspects can be automatically generated based on the requirements stated by components and applications, thus leveraging the component implementer of having to deal with these non-functional aspects. In this paper we present the characteristics and the convenience of the generated code for dealing with load balancing, distribution, and fault-tolerance aspects in the context of CORBA–$\mathcal{LC}$. CORBA–$\mathcal{LC}$ is a lightweight distributed reflective component model based on CORBA that imposes a peer network model in which the whole network acts as a repository for managing and assigning the whole set of resources: components, CPU cycles, memory, etc.[a]

1 Introduction

Component-based development (CBD)[1], resembling integrated circuits (IC) connections, promises developing application connecting independently-developed self-describing binary components. These components can be developed, built and shipped independently by third parties, and allow application builders to connect and use them. This development model is very convenient for distributed applications, as components can be installed in different hosts to match the distributed nature of this kind of application. Moreover, as applications become bigger, they must be modularly designed. Components come to mitigate this need, as they impose the development of modules that are interconnected to build complete applications. Components, being binary, independent and self-described, allow:

- Modular application development, which leads to maximum code reuse, as components are not tied to the application they are integrated in.

[a]This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants “Consolider Ingenio-2010 CSD2006-00046” and “TIN2006-15516-C04-03”.

- Soft application evolution and incremental enhancement, as enhanced versions of existing components can substitute previous versions seamlessly, provided that the new components offer the required functionality. New components can also be added to increase the set of services and functionality that new components can use, thus allowing applications to evolve easily.

To bring the benefits of Component-Based Development to distributed High Performance Computing (HPC), we developed CORBA *Lightweight Components* (CORBA–$\mathcal{LC}$)[2], a distributed component model based on CORBA[3]. CORBA–$\mathcal{LC}$ offers traditional component models advantages (modular applications development connecting binary interchangeable units), while performing an automatic deployment of components over the network. This deployment solves the component dependencies automatically, using the complete network of hosts to decide the placement of component instances in network nodes, intelligent component migration and load balancing, leading to maximum network resource utilization.

In order to perform this intelligent deployment, components separate the actual component functionality from the non-functional specification of Quality of Service (QoS) requirements, such as load balancing, fault tolerance, and distribution. This information is used by the CORBA–$\mathcal{LC}$ framework to generate the code that deals with those non-functional aspects of the component. In this way, the programmer can concentrate only on the component functionality, leaving to the framework the responsibility of ensuring that the actual QoS requirements are met. Moreover, separating component code from the specification of non-functional requirements allows us to apply *Aspect-Oriented Programming* (AOP)[4] techniques to the CORBA–$\mathcal{LC}$ Component Model. In this paper we show how AOP techniques can be used for automatic code generation of the aforementioned non-functional aspects code, and discuss the convenience of this approach of combining Component-Based Development (CBD) with AOP.

The paper is organized as follows: Section 2 offers an overview of CORBA–$\mathcal{LC}$. Section 3 shows how graphics applications can be used to define how to connect a set of components and how to specify non-functional aspects requirements. Section 4 shows how automatic code can be generated to seamlessly and transparently offer non-functional aspects implementation. Finally, Section 5 offers related work in the fields of component models and aspects, and Section 6 presents our conclusions.

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

CORBA Lightweight Components (CORBA–$\mathcal{LC}$)[2,5] is a lightweight component model based on CORBA, sharing many features with the CORBA Component Model (CCM)[6]. 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 produced and consumed, etc.) These are connection points with other components, called *ports*.

- *Containers and Component Framework*. Component instances are run within a run-time environment called *container*. Containers become the instances view of the

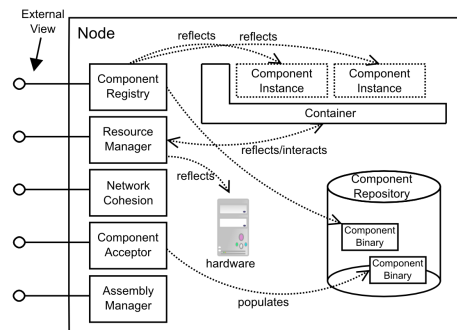

Figure 1. Logical Node Structure.

world. Instances ask the container for the required services and it in turn informs the instance of its environment (its *context*).

- *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 (*CORBA Interface Definition Language*) and XML files.

- *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 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 – Fig. 1. Nodes offer information about memory and CPU load, and the set of components installed.

  – *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) component instances running in the node and the properties of each, and (c) how those instances are connected via ports (assemblies); the *Resource Manager* in the node collaborates with the *Container* implementing initial placement of instances, migration/load balancing at run-time.

  – *Network Model and The Distributed Registry*. The CORBA–$\mathcal{LC}$ deployment model is a network-centred model: The complete network is considered as a repository for resolving component requirements.

  – *Applications and Assembly*. In CORBA–$\mathcal{LC}$, *applications* are a set of rules that a set of component instances must follow to perform a given task. Applications are also called *assemblies*, as they encapsulate explicit rules to connect component instances. Application deployment is then issued by instantiating an

assembly: creating component instances and connecting them. The CORBA–$\mathcal{LC}$ deployment process is intelligent enough to select the nodes to host the component instances based on the assembly requirements. Users can create assemblies using visual building tools, as the CORBA–$\mathcal{LC}$ Assembly GUI – Fig. 2.

3 Component Non-Functional Aspects

Components are not only a way of structuring programs, but a framework in which the programmer can focus in the functionality, leaving other aspects (called *non-functional aspects*) such as reliability, fault tolerance, distribution, persistence, or load balancing to the framework. The goal of CORBA–$\mathcal{LC}$ is to allow the programmer to write the functionality of the components, then describe how the components would work in terms of those non-functional aspects in a declarative manner, and let the framework to implement that requirements.

We can use an assembly of an example *Master/Worker* application based on components to show how CORBA–$\mathcal{LC}$ can achieve this goal. Figure 2 shows this assembly in the CORBA–$\mathcal{LC}$ Assembly GUI. The upper left part of the screen shows the available components in the network, while the lower left part shows the characteristics of the selected component or connection. Each red rectangle in the right part represents a component. Used interfaces are shown in the left part of the rectangle, while provided ones are shown in the right part. You can see the connection among components.

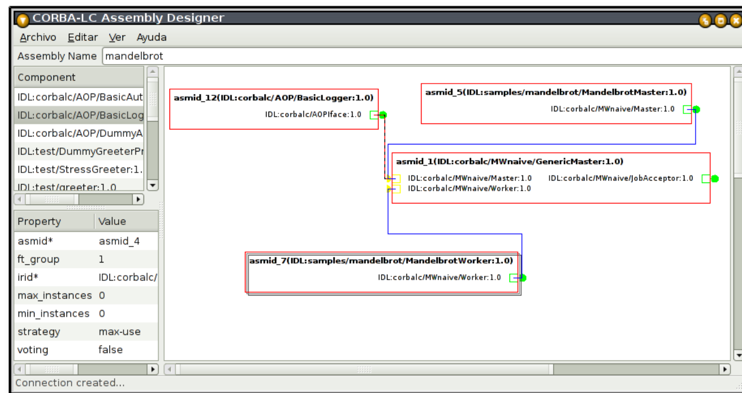

Figure 2. The CORBA–$\mathcal{LC}$ Assembler Graphical User Interface (GUI).

The `GenericMaster` component is in charge of the standard *Master/Worker* protocol. Note that the only needs of this component is to have one component that *provide* the generic `Master` interface, and a set of components that *provide* the generic `Worker` interface. This is very convenient, as allows us to plug *any* pair of components that perform that type of computation. In this case, the figure shows a pair of components that calculate a Mandelbrot fractal.

In the lower left part, the figure shows the set of properties of the `Worker` connection of the `GenericMaster` component instance (that is, the characteristics of the connection between the `GenericMaster` and the `MandelbrotWorker` component through the `Worker` interface.) The most important property is "`strategy`", which defines how this connection is made. In CORBA–$\mathcal{LC}$, this property can have different possible values:

- `default`. This is a normal CORBA call.
- `local`. The connection is instructed to be local (both components must be instantiated in the same node).
- `remote`. The connection is instructed to be remote (components must be instantiated in different nodes).
- `fault-tolerant`. With this value, instead of one instance of the component, a set of component instances (*replicas*) are created, in different nodes. Whenever the component calls an operation on this interface, several threads are created to issue concurrent calls to all the alive replicas, performing a final voting and a signalling of dead replicas if necessary.
- `load-balancing`. Similar to the previous one, instead of one component instance, several are created. When a call to that interface is issued, the call is redirected to the less loaded node.
- `max-use`. Again, instead of one instance, the whole network will be used to create as many component instances as possible.

The `local` strategy is useful for components that need an high-speed connection, such as streaming processing. The `remote` value instead is useful for keeping a small set of components in a low-end node (such as a PDA,) while the rest of components are in remote, more powerful, nodes. For the last three values, optional "`min_instances`" and "`max_instances`" properties can also be specified. When the CORBA–$\mathcal{LC}$ Assembler builds the application, it will ensure the number of needed component instances and connections satisfy the needs of the assembly. Note that in some cases, to meet the number required instances, this step may also require sending the component for installation and instantiation to other nodes if there are not enough nodes with this component available in the first place. In this specific example, the "`strategy`" property shows the "`max-use`" value, as suggested for the *Master/Worker* functionality.

Finally, AOP connections are also possible. In Fig. 2, the stripped connection line shows an AOP connection. These connections allows a component (that provides the `AOPIface` interface) to take control each time a call is made between two components (allow it, abort it, or even modify it.) In the figure, the `BasicLogger` component is in charge of writing a log of all the calls made. Note that this behaviour goes unnoticed for both the caller and the callee, and that this functionality can be activated or deactivated at any time, even at run-time, without any specific code written for logging in the application.

Other AOP connections can be made. In our research, an authenticator component has also been created, allowing any connection to be authenticated prior to making any call.

4 Automatic Generation of Aspects Code

The CORBA–$\mathcal{LC}$ *Code Generator* is in charge of generating all the code to deal with the required non-functional aspects of components. It uses the information of both (1) the

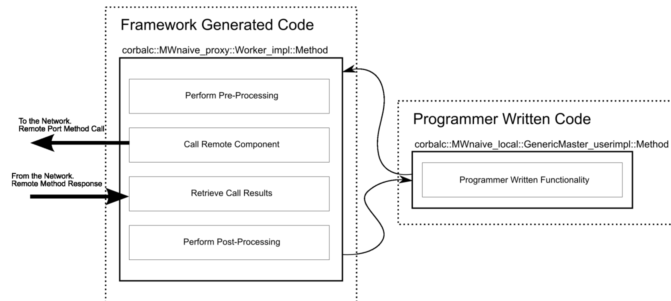

Figure 3. Proxy call sequence.

standard CORBA *Interface Repository* (IR), and (2) the Component's XML file. While the XML describes the component ports and non-functional requirements, the IR describes all the IDL interfaces used in those ports. As output, the code generator produces the needed implementation files and boilerplate that can be used by the programmer to write the functionality proper of the component.

CORBA–$\mathcal{LC}$ must have control of all the communication that happens among the component ports. This way, the framework can modify how the components communicate assuring the required load balancing, fault-tolerance, etc. Thus, for each used and provided interface, interception code must be created by the Code Generator. This is also referred to as "*point-cuts*" in Aspect-Oriented Programming (AOP)[4] terminology.

For each provided interface of a component, CORBA implementation objects (*servants*) are created. Servants are in charge of receiving the actual CORBA calls, and propagating the call to the final programmer code (called *executor*), that implements the functionality of the offered interface. The servant can also perform pre- and post-processing on the call. For instance, it can retrieve and store the executor data from a data-base, offering seamless persistence (as another aspect) to component instances.

For each required (*used*) interface of a component, *proxy objects* are created. They are in charge of delivering the local programmer code call to other component's provided interface as a normal CORBA call. At this point, the proxy code can also do some pre- and post-processing. Concretely, they are in charge of Fig. 3: (a) Calling the possibly attached AOP connections to this port, passing them all the parameters of the call, (b) Maintaining the set of active remote component instances, and (c) Depending on the *strategy*:

- Generating a pool of threads and concurrently calling all the remote component instances, retrieve their results and optionally performing voting (*fault-tolerant*.)
- Localizing the less loaded node and sending the call to the component instance running in that particular node (*load-balancing* strategy.)
- Providing to the component the set of remote component instances (*max-use*.)

5 Related Work

To date, several distributed component models have been developed. Although CORBA–$\mathcal{LC}$ shares some features with them, it also has some key differences.

Java Beans[7], Microsoft's Component Object Model (COM)[8], .NET[9] offer similar component models, but lack in some cases that are either limited to the local (non-distributed) case or do not support heterogeneous environments of mixed operating systems and programming languages as CORBA does.

In the server side, SUN's EJB[10] and the recent Object Management Group's CORBA Component Model (CCM)[11] offer a server programming framework in which server components can be installed, instantiated and run. Both are fairly similar. Both are designed to support enterprise applications, offering a container architecture with support for transactions, persistence, security, etc. They also offer the notion of components as binary units which can be installed and executed (following a fixed assembly) in Components Servers.

Although CORBA–$\mathcal{LC}$ shares many features with both models, it presents a more dynamic model in which the deployment is not fixed and is performed at run-time using the dynamic system data offered by the Reflection Architecture. Also, CORBA–$\mathcal{LC}$ is a *lightweight* model in which the main goal is the optimal network resource utilization instead of being oriented to enterprise applications. Finally, CORBA–$\mathcal{LC}$ adds AOP connections, not present in the other two models.

Applying Aspects-Oriented techniques to Component Models has also been explored in several works. In[12] the authors apply AOP techniques to the EJB component model. This work is limited to Java and the usage of AspectJ[13] to provide a finer grain of control over actual calls in EJB. A quantitative study showing the benefits of AOP for component-based applications (in terms of number of lines of code and number of places to change when a modification on the application has to be done) can be found in[14].

In[15], the authors apply aspect oriented techniques in the context of the CORBA Component Model and security policies using the Qedo framework (an implementation of the CCM). Real-Time has been treated as an aspect in a CCM component framework implementation (CIAO) in[16]. The approach of these works is similar to the one presented in this paper, but none of them treat distribution, load balancing and fault tolerance as an aspect.

In the field of High Performance Computing (HPC) and Grid Computing, Forkert et al. ([17]) present the TENT framework for wrapping applications as components. However, this wrapping is only used to better organize applications, and not to provide an integrated framework in which offer services to component implementations.

The Common Component Architecture (CCA)[18] is a component model framework also based on the idea of reusable, independent components. However, it does not offer any basic run-time support for distribution, load balancing or fault tolerance. Thus, implementing those services require of *ad-hoc* programming, which goes against reusability.

6 Conclusions

As we showed in this paper, component technology in general, and CORBA–$\mathcal{LC}$ in particular, offer a new and interesting way of approaching distributed applications. Services otherwise complicated can be offered by the framework just by specifying them in the characteristics and needs of components and applications. We showed how convenient the Aspect-Oriented approach is to seamlessly and transparently offer services such as fault tolerance, replication and load balancing to components, and the importance of being able to specify those non-functional aspects in a declarative manner, so that the required code for those aspects can be generated automatically.

References

1. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, (ACM Press, 1998).
2. D. Sevilla, J. M. García and A. Gómez, *CORBA lightweight components: a model for distributed component-based heterogeneous computation*, in: EUROPAR'2001, LNCS vol. **2150**, pp. 845–854, Manchester, UK, (2001),
3. M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*, (Addison-Wesley Longman, 1999).
4. F. Duclos, J. Estublier and P. Morat, *Describing and using bon functional aspects in component-based applications*, in: International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, (2002).
5. D. Sevilla, J. M. García and A. Gómez, *Design and implementation requirements for CORBA Lightweight Components*, in: Metacomputing Systems and Applications Workshop (MSA'01), pp. 213–218, Valencia, Spain, (2001).
6. Object Management Group, *CORBA: Common Object Request Broker Architecture Specification, revision 3.0.2*, (2002), OMG Document formal/02-12-06.
7. SUN Microsystems, *Java Beans specification*, 1.0.1 edition, (1997). `http://java.sun.com/beans`.
8. Microsoft, *Component Object Model (COM)*, (1995). `http://www.microsoft.com/com`.
9. Microsoft Corporation, Microsoft .NET, `http://www.microsoft.com/net`
10. SUN Microsystems, *Enterprise Java Beans specification*, 3.0 edition, May 2006, `http://java.sun.com/products/ejb`.
11. Object Management Group, *CORBA Component Model*, 1999, OMG Document ptc/99-10-04.
12. R. Pichler, K. Ostermann and M. Mezini, *On aspectualizing component models*, Software, Practice and Experience, **33**, 957–974, (2003).
13. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science, **2072**, 327–355, (2001).
14. O. Papapetrou and G. Papadopoulos, *Aspect oriented programming for a component based real life application: a case study*, in: Symposium on Applied Computing — Software Engineering track, (2004).
15. T. Ritter, U. Lang and R. Schreiner, *Integrating security policies via container portable interceptors*, Distributed Systems Online, (2006).
16. N. Wang, C. Gill, D. C. Schmidt and V. Subramonian, *Configuring real-time aspects in component middleware*, in: International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus, (2004).
17. T. Forkert, G. K. Kloss, C. Krause and A. Schreiber, *Techniques for wrapping scientific applications to CORBA Components.*, in: High-Level Parallel Programming Models and Supportive Environments (HIPS'04), pp. 100–108, (2004).
18. D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan and A. Slominski, *On building parallel & grid applications: component technology and distributed services.*, Cluster Computing, **8**, 271–277, (2005).