Aspect-Oriented Programing Techniques to support Distribution, Fault Tolerance, and Load Balancing in the CORBA–$\mathcal{LC}$ Component Model*

Diego Sevilla, José M. García
Computer Engineering
University of Murcia, Spain
30071 Murcia. Tel: +34968367571
{dsevilla, jmgarcia}@ditec.um.es

Antonio Gómez
Information and Communications Engineering
University of Murcia, Spain
skarmeta@dif.um.es

Abstract

The design and implementation of distributed High Performance Computing (HPC) applications is becoming harder as the scale and number of distributed resources and application is growing. Programming abstractions, libraries and frameworks are needed to better overcome that complexity. 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.

*This work has been supported by the Ministry of Education and Science of Spain under grants TIN2006-15516-C04-03 and CSD2006-00046.

1. Introduction

Component-based development (CBD)[22], 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, matching the physically distributed nature of this kind of applications.

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.
- 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–$\mathcal{LC}$)[18], a distributed component model based on CORBA[6]. 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)[2] 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–$\mathcal{LC}$) [18, 19] is a lightweight component model based on CORBA, sharing many features with the CORBA Component Model (CCM)[12]. 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 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.

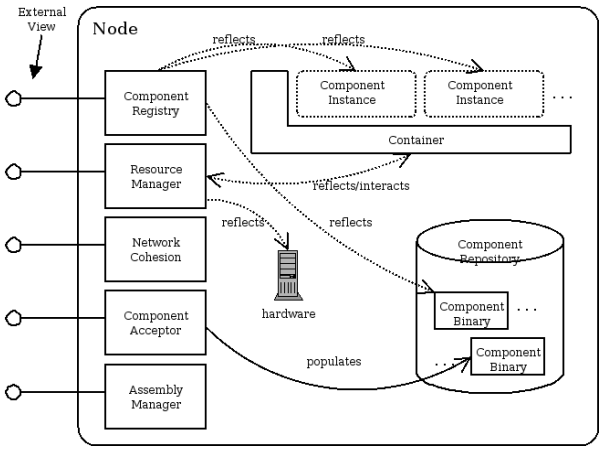

Figure 1. Logical Node Structure.

- **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. The packaging allows storing different binaries of the same component to match different Hardware/Operating System/ORB. The package can be used to send components to install in a desired node, even at run-time.

- **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 (Assemblies).

  - **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, as well as 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) the set of component instances running in the node and the properties of each, and (c) how those instances are connected via ports (assemblies)[14]. This information is used when components, applications or visual builder tools need to obtain information about components.
  - 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-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.

– **Applications and Assembly**. In CORBA–$\mathcal{LC}$, *applications* are a set of rules that a set of components and component instances must follow to perform a given work. 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. Given the distributed nature of CORBA–$\mathcal{LC}$, the 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 Designer Graphical User Interface (Fig. 2).

This is the framework we will use to study the implications and suitability of introducing Aspect-Oriented techniques to Component-Based development in the domain of High-Performance Computing.

3. Specifying Non-Functional Aspects for Components

As stated before, components are not only a way of structuring programs, but a framework in which the programmer can focus in the functionality, leaving other 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.

Thus, when a programmer wants to write an application with CORBA–$\mathcal{LC}$, he or she defines the different components that will take part into the application, designing their interfaces and establishing the set of used and provided interfaces and events of each component. This results in IDL and XML files describing each component. These files are used by the CORBA–$\mathcal{LC}$ Code Generator (described in §4) to generate the code that allows that component to deal with non-functional aspects. The generated code also includes boilerplate and hooks in which the programmer can write the actual component implementation, without having to deal with non-functional aspects.

Once the components are complete and packaged, the whole application has to be designed, normally using a Graphical User Interface (GUI) similar to the CORBA–$\mathcal{LC}$ Assembly Designer GUI shown in Figure 2. This means selecting the components that will take part into the application, specifying the connections between used and provided interfaces of each component, as well as connections of events emitted and received by each component. It is at this point in which the programmer has to decide how those connections will behave in terms of load balancing, network usage and fault tolerance. An assembly XML file with that information is created. When the programmer wants to start the application, feeds this file to the CORBA–$\mathcal{LC}$ Assembler, which is in charge of actually finding the components installed on network nodes, creating the needed component instances, connecting them, and configuring the connections as specified in the assembly.

We can use an example assembly of a *Master/Worker* application based on components to show how to specify these non-functional aspects. Figure 2 shows this assembly in the CORBA–$\mathcal{LC}$ Assembly Designer 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 rectangle in the right part of the figure (also called the *design area*) represents one or more component instances. Figure 3 shows the GUI representation of the `GenericMaster` component. The set of used interfaces are displayed in the left side of the rectangle (as a half yellow circle), while provided interfaces are shown in the right side (as a green full circle). `GenericMaster` uses both the `Master` and `Worker` interfaces, and offer the `JobAcceptor` so that clients can send calculations to be

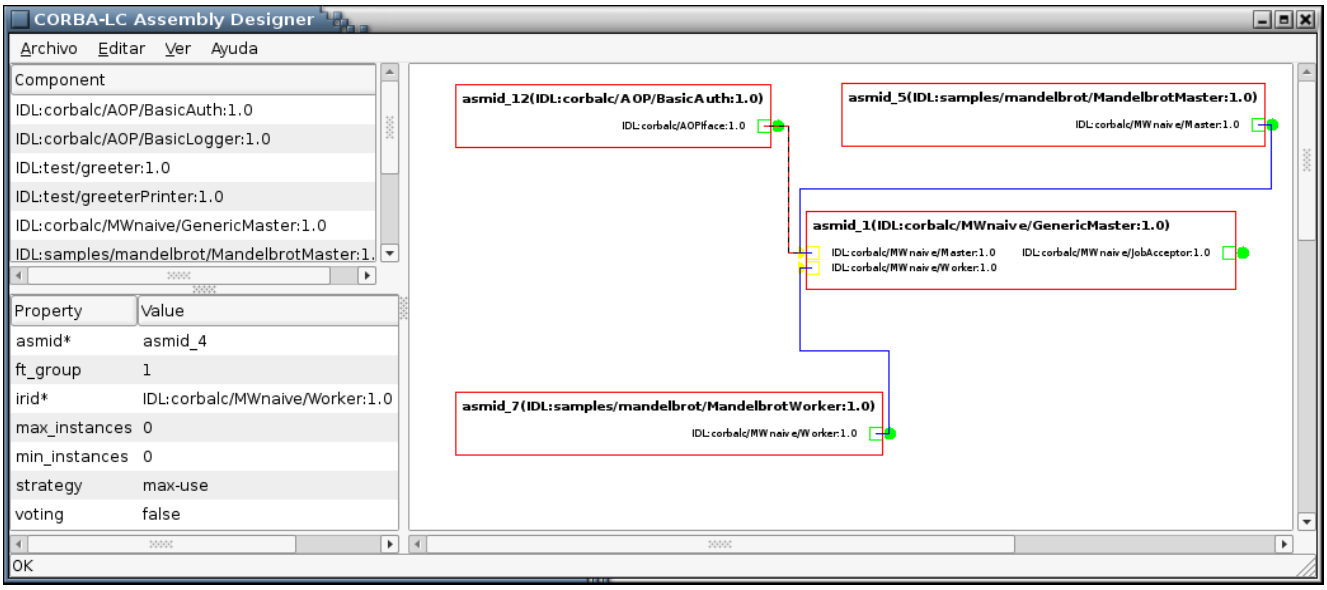

Figure 2. The CORBA–$\mathcal{LC}$ Assembly Designer GUI.

performed. These communication ports are *synchronous*, and are equivalent to normal CORBA interfaces. Although this particular component only has these synchronous communication ports, in CORBA–$\mathcal{LC}$, components also support *asynchronous* communication endpoints, known as *events*. A component can declare that it emits or receives a set of events.

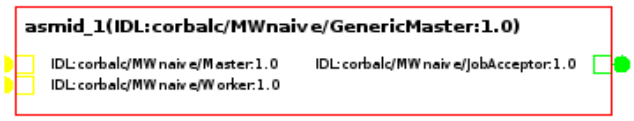

Figure 3. The `GenericMaster` CORBA–$\mathcal{LC}$ Component.

Connection among the components ports are also drawn (in blue). Note that both the provided and used interfaces of each connection are of the same interface type, described by its CORBA Interface Repository Identifier. These interfaces are described in the IDL files that are included within the component package.

The `GenericMaster` component is in charge of the standard *Master/Worker* protocol. Note that the only requirements of this component in terms of connections 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 example, the figure shows a pair of components that calculate a Mandelbrot fractal.

In the lower left panel, Figure 2 shows the set of properties of the component or connection selected in the design area. In this example, we selected the `Worker` connection of the `GenericMaster` component instance. That is, the properties of the connection between the `GenericMaster` and the `MandelbrotWorker` components through the `Worker` interface are shown. Of the properties shown, the most important property is "`strategy`", which defines the *behavioral characteristics* of this connection. In CORBA–$\mathcal{LC}$, this property can hold different possible values:

- `default`. This is a normal call. Components are described as using and providing a set of CORBA interfaces. This call is then equivalent to a standard synchronous CORBA call. The client component calls operations in the interface, which are implemented by the provider component.
- `local`. The deployer is instructed to allocate the instances of the caller and called component in the same host. This is needed for some high-speed connections (for instance, video or audio streaming, compression, filtering, etc.)
- `fault-tolerant`. Specifying this value for the strategy of a connection instructs the deployment process to create a set of component instances (*replicas*), in different nodes. Whenever the client component calls an operation on this interface (that is, makes use

of the connection), several threads are created to issue concurrent calls to all the alive replicas. Finally, if the value of the "voting" property is set to "true", a final voting process is issued, signaling of dead/faulting replicas if necessary.

- load-balancing. Similarly to the previous one, instead of one component instance, several are created. When the client component (that was described as "using" this interface) calls to that interface, the call is redirected to node with lower CPU load.

- max-use. Again, instead of one component instance that provides an implementation for the interface, the whole network is used to create as many component instances as possible, in different nodes, to maximize resource usage.

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 requirements specified in the assembly. Note that in some cases, to meet the number required instances, this step may also require sending the component binary to other nodes for installation and instantiation, if there are not enough nodes with this component available in the first place.

Finally, load-balancing and max-use strategies can be tailored adding a level of fault tolerance by specifying a value of the "ft_group" property. This property defines the size of the group of instances that will be treated as a fault-tolerant domain. That is, instead of creating n instances between min_instances and max_instances, a number of n *groups* (of ft_group instances each) are created. Instances within the group are considered as fault-tolerant replicas, and the same tests and voting is performed as in the fault-tolerant strategy above, but within each group (Figure 4). When the value of this property is 1, the property has no special meaning.

It is clear that in this specific example, we can set the "strategy" property on the Worker used interface of the GenericMaster component to the "max-use" value, as suggested for the Master/Worker functionality. Thus, as many as possible MandelbrotWorker components are supplied to the GenericMaster component, all of them connected through the Worker used port. Optionally, we can specify a number in the "ft_group" property if we want each instance to be replicated.

Finally, AOP connections are also possible. In Figure 2, the stripped red connection line shows an AOP connection. These connections allow a component (that provides the AOPIface interface) to take control each time a call is made between two components through that connection.

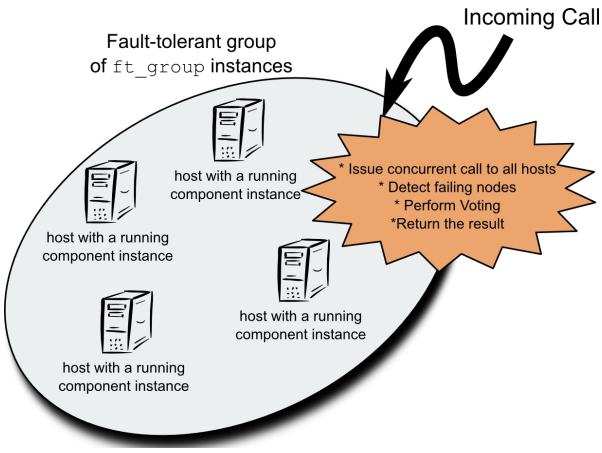

Figure 4. A group of instances running on hosts that acts as a fault-tolerant group.

The AOP component then can allow the call, abort it, or even modify the value of parameters.

```
module corbalc
{
        typedef sequence<any> AnySeq;

        // AOP Iface definition.
        interface AOPIface : Iface
        {
                boolean pre (in string
                    iface, in string opname,
                        inout AnySeq
                            params);

                boolean post (in string
                    iface, in string opname,
                        inout AnySeq
                            params);
        };
};
```

Figure 5. The AOPIface IDL interface.

Figure 5 shows the AOPIface IDL interface. The pre and post methods are called before and after each call. The set of parameters of the call are passed to the method using an inout parameter. This way, the AOP component can change the values of the parameters. Finally, each method returns a boolean value. A true return value will allow the call.

In the example, we used the BasicLogger component between the GenericMaster and the MandelbrotWorker components. This component simply writes to standard output a log of all the calls made

through that interface connection (logging the calls made from the generic master to the worker component). This kind of AOP connections are very convenient, as:

- The call interception goes unnoticed for both the caller and the callee.
- New AOP connections can be made at run time, activated, and deactivated, being able to control every connection among components. In the example, switch on and off the logging.
- No code has to be written in any component to deal with logging of calls.
- Other AOP components can be created. For example, in our research, we created an authenticator component, that allows any connection to be authenticated prior to making any call.

Although CORBA–$\mathcal{LC}$ is not restricted to Master/Worker type of applications, with this example we showed how convenient mixing component technology and the specification of non-functional aspects is for High-Performance Computing Applications.

In the next section we show how the code generated by the CORBA–$\mathcal{LC}$ Code Generator can deal with all these specified aspects and AOP connections.

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 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. All this code is compiled, bundled jointly with the XML and IDL files describing the component, and compressed into a component package, ready to be installed and instantiated in hosts.

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, code that intercepts that communication must be created by the Code Generator. This is also referred to as "*point-cuts*" in Aspect-Oriented Programming[2] 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 (Fig. 6).

Similarly, for each required (*used*) interface of a component, *proxy objects*[4] are generated. Proxy objects are local representatives of the actual used port of the remote component. They are in charge of delivering the 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, as shown in Figure 7. Concretely, for each method of the used interface, proxy code is in charge of:

- Calling the possibly attached AOP connections to this port, passing them all the parameters of the call.
- Maintaining the set of active remote component instances.
- Depending on the selected *strategy* for this particular connection:
  - Generating a pool of threads that concurrently call all the remote component instances, retrieve their results and optionally perform some form of voting (*fault-tolerant* strategy.)
  - Locating the less loaded node and sending the call to the component instance running in that particular node (*load-balancing* strategy.)
  - Providing the component the set of remove component instances (*max-use* strategy.)

The generated code for all the proxy objects and the servants for a component is included in the component binary, jointly with the programmer written functionality for that component. When the CORBA–$\mathcal{LC}$ assembler builds the specified application (for example, that appearing in Figure 2), it instantiates all the components in their corresponding nodes and configures the generated code to work as specified in the assembly. From that point on, the generated code is in charge of interacting with the container to fulfill the desired load balancing and fault tolerance requirements.

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[20], Microsoft's Component Object Model (COM)[8], .NET[1] offer similar component models, but

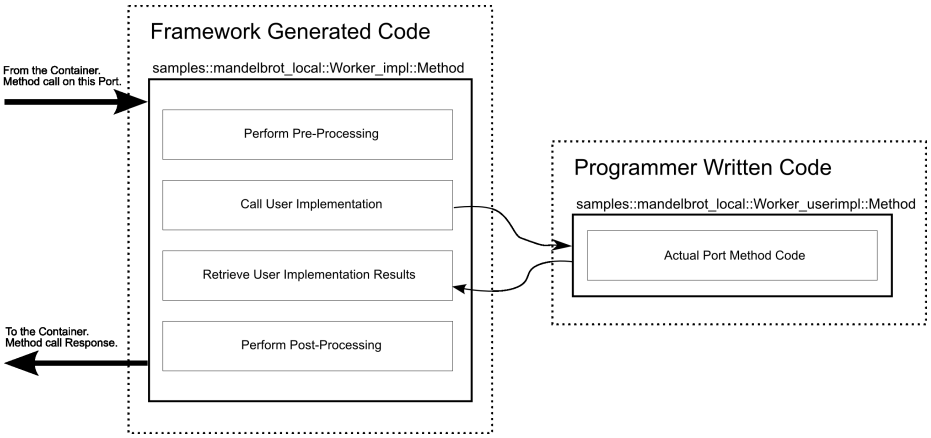

Figure 6. Executor call sequence.

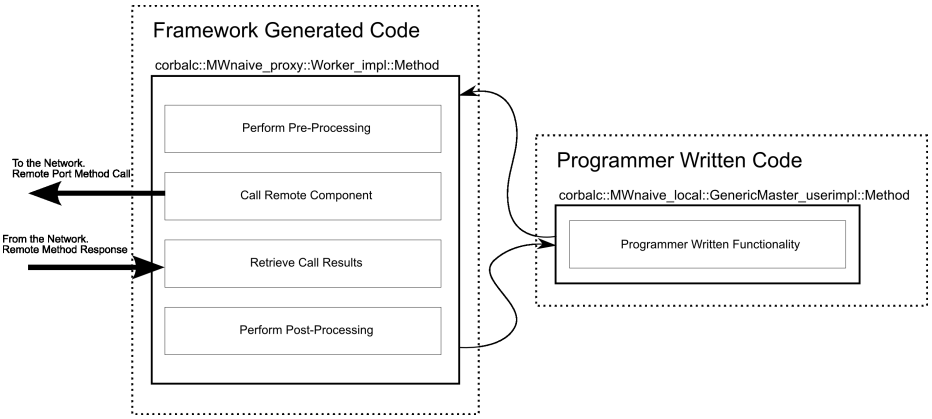

Figure 7. Proxy call sequence.

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[21] 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 [15] the authors apply AOP techniques to the EJB component model. This work is limited to Java and the usage of AspectJ[7] 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 [13].

For the implementation of fault tolerance in the CORBA environment, we can use works in the Eternal System[10] and Group Multicast for CORBA[9]. However, we are more interested in building an architecture and design model that allows a seamless integration of those techniques with the complete component specification, design and interaction, as shown in the application assembly specification example.

In [16], 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 implementaion (CIAO[17]) in[23]. 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. ([3]) 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)[5] 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

Component technology in general, and CORBA–$\mathcal{LC}$ in particular, offers a new and interesting way of approaching distributed applications. Services otherwise complicated can be offered by the framework 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] M. Corporation. Microsoft .NET. http://www.microsoft.com/net/.

[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] 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)*, pages 100–108, 2004.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] 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(4):271–277, 2005.

[6] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley Longman, 1999.

[7] 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.

[8] Microsoft. *Component Object Model (COM)*, 1995. http://www.microsoft.com/com.

[9] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, R. R. Koch, and K. Berket. Multicast Group Communication for CORBA. In *International Symposium on Distributed Objects and Applications*, pages 98–107, 1999.

[10] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal – A Component-based Framework for Transparent Fault-Tolerant CORBA. *Software Practice and Experience*, 32(8):771–788, July 2002.

[11] Object Management Group. *CORBA Component Model*, 1999. OMG Document ptc/99-10-04.

[12] Object Management Group. *CORBA: Common Object Request Broker Architecture Specification, revision 3.0.2*, 2002. OMG Document formal/02-12-06.

[13] 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.

[14] 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.

[15] R. Pichler, K. Ostermann, and M. Mezini. On aspectualizing component models. *Software, Practice and Experience*, 33(10):957–974, 2003.

[16] T. Ritter, U. Lang, and R. Schreiner. Integrating security policies via container portable interceptors. *Distributed Systems Online*, July 2006.

[17] D. C. Schmidt. Component-Integrated ACE ORB (CIAO), 2006. `http://www.cs.wustl.edu/~schmidt/CIAO.html`.

[18] D. Sevilla, J. M. García, and A. Gómez. CORBA Lightweight Components: A Model for Distributed Component-Based Heterogeneous Computation. In *EUROPAR'2001*, pages 845–854, Manchester, UK, August 2001. LNCS 2150.

[19] D. Sevilla, J. M. García, and A. Gómez. Design and Implementation Requirements for CORBA Lightweight Components. In *Metacomputing Systems and Applications Workshop (MSA'01)*, pages 213–218, Valencia, Spain, September 2001.

[20] SUN Microsystems. *Java Beans specification*, 1.0.1 edition, July 1997. http://java.sun.com/beans.

[21] SUN Microsystems. *Enterprise Java Beans specification*, 3.0 edition, May 2006. http://java.sun.com/products/ejb.

[22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1998.

[23] 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, October 2004.