Design and Implementation Requirements for *CORBA Lightweight Components*[*]

Diego Sevilla, **José M. García**
Dept. of Computer Engineering
University of Murcia, Spain
{dsevilla,jmgarcia}@ditec.um.es

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

Abstract

This paper describes the guidelines we followed and the requirements we stated while designing and implementing the CORBA Lightweight Components *(CORBA–$\mathcal{LC}$) Component Model. CORBA–$\mathcal{LC}$ is a lightweight distributed reflective component model based on CORBA. Instead of using a traditional client/server model, it imposes a peer network model in which the whole network act as a repository for managing and assigning the whole set of resources: components, CPU cycles, memory, etc. Thus, application deployment is automatically and adaptively performed at run-time. Requirements for component description, packaging, deployment, reflection, logical network cohesion, distributed resource queries and fault-tolerant protocols are identified. Finally, we show the validity of the identified requirements in dealing with CSCW and Grid Computing applications and show how current component models fail on addressing some of these requirements.*

1. Introduction

Component-based development [32, 33], 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.

As application 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 the complete application. 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 add new functionality to be used by new components, thus allowing applications to evolve easily.

When component technology is applied in a distributed environment, programmers can develop components that interact transparently with other components residing in remote machines. However this makes applications and components management harder.

CORBA *Lightweight Components* (CORBA–$\mathcal{LC}$) [29], is a new distributed component model based on CORBA [11, 19]. While traditional component models force programmers to decide the hosts in which their components are going to be run (deployment) using a "static" description of the application (*assembly*), CORBA–$\mathcal{LC}$ performs the deployment and component dependency management automatically. Thus, it offers the traditional component models advantages (modular applications development connecting binary interchangeable units) allowing automatic placement of components in network nodes, intelligent component migration and load balancing, leading to maximum network resource utilization. CORBA–$\mathcal{LC}$ introduces a more *peer* network-centered model in which all node resources, computing power and components can be used at run-time to automatically satisfy applications dependencies.

While CORBA–$\mathcal{LC}$ is described elsewhere [29], this paper focuses on the requirements the design and implementation of the CORBA–$\mathcal{LC}$ component model must address to cope with the demands of CSCW and Grid computing applications. The paper is organized as follows: Section 2 describes the general design and implementation guidelines of CORBA–$\mathcal{LC}$. Section 3 outlines the needs of both CSCW and Grid applications and how CORBA–$\mathcal{LC}$ addresses them. Section 4 includes some related work on general purpose component models and how CORBA–$\mathcal{LC}$ relates to it. Finally, Section 5 presents our conclusions, the status of current implementation and the expected future work.

[*] This work is partially supported by the Spanish SENECA Foundation under Grant PB/13/FS/99.

2. Design and implementation guidelines for CORBA–$\mathcal{LC}$

While designing the CORBA–$\mathcal{LC}$ Component Model [29], we identified some design and implementation guidelines and requirements. These requirements state what we demand from component-based development in a distributed environment, and include some key features we miss in current component models. We were interested in Computer Supported Cooperative Work (CSCW) and Grid computing, so we designed CORBA–$\mathcal{LC}$ for dealing with these. Requirements identified include:

1. **Simplicity and performance**. The model should be simple enough to accommodate component-based development and to allow being implemented efficiently. In other words, it must be *lightweight*.

2. **Heterogeneity**. It must support heterogeneous resource integration at any level: language, hardware, operating system and network. CORBA [19, 11] reveals here to be a perfect election.

3. **Peer** or **Network-Centered Model**. We are interested in integrating the whole computation power available, including distributed network-connected machines, so all the nodes connected must collaborate as peers depending on their available resources. The traditional client and server distinction is not applicable, as nodes may change their role as needed. Thus, the network (with help of nodes composing it) becomes the logical entity in charge of maintaining and administering the available resources (components, CPU cycles, etc.) of the whole network.

4. **Scalable and fault-tolerant**. In order to accommodate a potentially large number of hosts in a distributed environment, the need for distributed scalable and fault-tolerant protocols arise. These protocols essential to support the network behavior.

5. **Seamlessly integrate new components**. It must be possible to add new components into the system (without the need of compiling) and make them instantly available to be used by *any* application in *any* host.

6. **Automatic component dependency management**. In a distributed system with hundreds or thousands of components, managing component interdependencies is not an easy task. New components installed in a host may require other components or new version of existing components not present in that host. Instead of forcing all required components to be installed in a host, the network as a whole must be used as a repository for resolving component requirements, fetching them from the host they are installed or using them remotely.

7. **Use the *same* component model for all application tiers**. Particularly, for CSCW applications, it would be desirable to include the GUI and presentation logic tiers components within the same design process of the rest of the application tiers.

8. **Integration of tiny devices**. The resource utilization logic must be intelligent enough to accommodate tiny devices such as PDAs as well as high-end servers.

2.1. Components

Components are the key abstraction in CORBA–$\mathcal{LC}$. Although there is some controversy on *what is a component* [33], in part because the term "component" being overutilized, we define them in the CORBA–$\mathcal{LC}$ model as *binary independent units, with explicitly defined dependencies and offerings, which can be used to compose applications*. This *natural* definition of components allows us to state the requirements we impose to the whole idea of Component-based Development [32] from a practical point of view.

Components must allow application development by assembling independent components. Components could be developed, described and packaged by independent parties and must interact seamlessly with other components.

These two properties (component independence and explicit dependencies) allow components to be substituted by others with the same (or even superior) offerings but with enhanced implementation or Quality of Service (QoS) guarantees. Substitutability also enables soft application migration, as they can be enhanced incrementally.

For this to happen, components must describe their interaction with the rest of components, in form of what they demand from the system and what they offer to it. Components also inherit from the traditional Object-oriented paradigm the fact that they are also a description of the run-time behavior and requirements of their *instances*. The instances then become running representations of the code stored in a component. Component descriptions then must satisfy at least two dimensions: the *binary package* or *static* dimension and the *component type* or *dynamic* dimension.

2.1.1. Binary package

Static properties include those referring to the binary package in which the component is shipped. This information is necessary to handle, store and manipulate this component in binary form. Concretely, components must include information which allows them:

- To be installed in a given host

- To be extracted from, and brought to, a given host
- To be dynamically loaded and unloaded as a Dynamic Link Library (DLL) either on client's behalf or attending to memory, CPU or bandwidth load.
- To be instantiated.

Thus, static properties description must include, at least [13]:

- Hardware, Operating System and Object Request Broker dependencies.
- Other components needed.
- Static description of offerings and needs, including:
  - **Mobility**: if the component can be extracted from a given host or it must be used remotely from this location.
  - **Replication**: if component instances can be replicated, either because they are stateless or they know how interact with the framework to maintain replica consistency.
  - **Aggregation** [17]: if this component knows how to split itself in different instances to process a set of data (data-parallel components) and how to gather partial results into a complete solution.
  - **Pay-per-use information**: describes the licensing model for this component.
  - **Security information**: The installer must be sure of who really made this component by verifying the component's cryptographic signature, for example, from the component's writer Web site.

This information is described using XML files for convenience, and stored in the package jointly with the component binary, as described in §2.3. The *Document Type Definitions* (DTDs) describing those files are based upon the WWW Consortium's *Open Software Descriptor* [35] DTD[1].

2.1.2. Component Type

Component instances are run-time incarnations of the behavior stored in a component. Thus, components are also a description of run-time properties and requirements of their instances. This description can be considered as the ***dynamic*** dimension. These properties and requirements can be *internal* or *external*, and allow components instances to be interconnected together to perform the required applications behavior.

Internal properties are those that instances expose to the framework in which they are immerse. Components must be executed in a controlled way, so they have to follow an agreed protocol to specify which are their run-time environment requirements and what they offer to the environment: they have to follow the framework rules. This framework is described in the following subsections.

External properties are those services that component instances expose to their clients (including other components or applications). Those external communication points are collectively called ***ports***. Ports, as happens in both the CCM [18] and CCA [1] models, allow components to be connected together to accomplish the required task.

CORBA–$\mathcal{LC}$ does not limit the different port kinds that a component can expose. However, there are two basic kinds of ports: *interfaces* and *events*.

A component can indicate that its instances implement (***provide***) or ***use*** some interfaces[2] for their internal work. Interfaces represent agreed synchronous communication points between components.

Events can be used as asynchronous communication means for components. They can also specify that they produce or consume some kind of event in a publish/subscribe fashion. For each event kind produced by a component, the framework opens a push event channel. Components can subscribe to this channel to express its interest in the event kind produced by the component.

Finally, factory interfaces [7] are needed in CORBA–$\mathcal{LC}$ to manage the set of instances of a component. Clients can search for a factory of the required component and ask it for the creation of a component instance.

Similar to static properties, components use a set of IDL and XML files to establish the *minimal* set of ports they need from and offer to other components. Those files are included within the component binary package (§2.3).

Concretely, component ***internal properties*** description must include:

- **Factory properties**: A description of the life cycle of the instances of the component, which allows to automatically generate the factory code for this type of component.
- **Required framework services**.
- **QoS information**: such as minimum and maximum CPU and memory utilization and minimum communication bandwidth needed, etc.

Similarly, component ***external properties*** description must include, at least:

- **IDL types and interfaces**: definition of types and interfaces related to the component.

[1] As the CORBA Component Model [18] (CCM) does.

[2] We use "interface" here in the same sense that it is used in CORBA.

- **Ports descriptions**: components must state the minimal set of ports (interfaces, events, etc.) offered and required.

- **Factory interface**: CORBA interface which implements the *factory pattern* [7] for this component, used for the creation of running instances.

Note that components only state their ***minimal*** requirements. This is because CORBA–$\mathcal{LC}$ does not restrict the set of external properties of a component to be fixed and allows it to change at run-time, offering new services and requesting new ones. This is supported by the reflection architecture described in §2.4.2.

In CORBA–$\mathcal{LC}$, we have chosen to use IDL files for specifying component's types and interfaces and XML files (instead of a modified IDL+CIDL combination, as proposed by the CCM) with a custom DTD to specify component-related external properties. This allows us to use CORBA 2 standard, mature IDL compilers and tools while obtaining the benefits of components. Moreover, it allows the seamless integration with current CORBA 2 software and a soft migration to component technology.

2.2. Containers and Component Frameworks

Component instances are run within a run-time environment called a ***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*). As in CCM and EJB, in CORBA–$\mathcal{LC}$ the component/container dialog is based on agreed local interfaces, thus conforming a component framework. Containers leverage the component implementation of dealing with the non-functional aspects of the component [4], such as instance activation/de-activation, resource discovery and allocation, component migration and replication, load balancing [24] and fault tolerance among others. For example, when it is determined[3] that some component instance should be run in another host of the network, the container can ask the component instance (via local agreed interfaces) to resume its execution returning its internal state. Then, the component can be migrated into another host (in its binary form), instantiated, and then given the previous instance state to continue its execution.

Containers also act as component instance representatives into the network. As stated above, components require other components to perform their work. This requirement is sent to the container and may require searching the complete network of cooperating nodes for the best suited component. The container collaborates with its local node in order to find the required component following the deployment and reflection architecture described in §2.4.

[3]This determination can be taken by the container in collaboration with the network.

2.3. Packaging

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. This is similar to the CCM Packaging Model [18]. This information can be used by each node in the system to know how to install and instantiate the component.

The requirements we impose to the component packaging include:

- It must include both the binary information and the *meta*-information for this component, including the DLLs (or *shared objects*, in the case of Unixes) and the IDL and XML files described above.

- It must admit compression to overcome the efficient transmission of the component through possibly long and slow communication lines.

- Must be modular enough to allow (1) storing binaries for different architectures/operating systems/ORBs, (2) describing those binaries, and (3) extracting only a set of binaries from the whole component (jointly with the component metadata) to be installed in devices with a tiny memory, such as PDAs. For instance, the same component could be implemented using a Windows DLL, a Java `.class` file, and a TCL script source code.

2.4. Deployment 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 the required applications behavior. 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**.

2.4.1. Nodes

The CORBA–$\mathcal{LC}$ network model can be effectively represented as a set of nodes that collaborate in computations. ***Nodes*** are the entities maintaining the logical network behavior. Each host participating must have running a server implementing the *Node* service. 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):

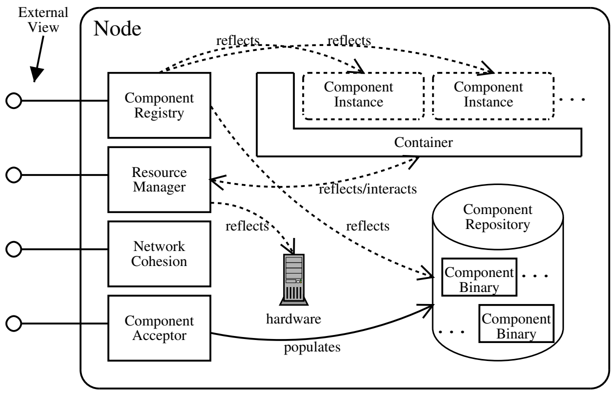

Figure 1. Logical Internal Node Structure.

- A way of obtaining both node static characteristics (such as CPU and Operating System Type, ORB) and dynamic system information (such as CPU and memory load, available resources, etc.) This is supported through the ***Resource Manager*** interface.
- A way of obtaining the external view of the local services: the ***Component Registry*** interface reflects the internal ***Component Repository*** and helps in performing distributed component queries.
- Hooks for accepting new components at run-time for local installation in the local Component Repository, instantiation and running [15]. This is performed using the ***Component Acceptor*** interface.
- Operations for making this node available to the network and to interact with the rest of nodes of the whole system. The ***Network Cohesion*** interface supports this protocol for logical network cohesion.

2.4.2. Reflection Architecture

The ***Reflection Architecture*** is composed of the meta-data given by the different node services. This meta-data reflects the node internal properties, both static and dynamic, and is used at various stages in CORBA–$\mathcal{LC}$. The Distributed Registry (§2.4.3) uses this information of each node to maintain an updated view of the resources of the whole network. Concretely, (Fig. 1):

- The ***Component Registry*** provides information about (a) the set of installed 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) [25]. This information is used:
  - when components or applications raise the need for either *any* instance or a *named* instance of a component,
  - by visual builder tools to offer to the user the palette of available components, instances and connections among them.
- The ***Resource Manager*** in the node collaborates with the Container in deciding initial placement of component instances and instance migration, replication and load balancing at run-time. The Resource Manager also reflects the hardware static characteristics and hardware resources dynamic usage and availability. This information is also used to determine if a component, depending on its hardware requirements, can be physically installed in the node.

With the help of the reflection architecture, new components (or new versions of existing components) can be aggregated to the system at any time, and become instantly and automatically available to be used by other components.

In contrast to CCM, the set of external properties of a component is not fixed and may change at run-time. Thus, component instances can adapt to the changing environment requesting new services or offering new ones. CORBA–$\mathcal{LC}$ offers operations which allow modifying the set of ports a component exposes [27].

2.4.3. 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. All hosts (nodes) in the system maintain a set of installed components in its Component Repository. All of those are available to be used by any other component. When component instances start running, they may ask their container for some required components. These components are searched in the whole network. The network issues the corresponding distributed queries to each node's Component Registry in order to find the component which match better with the stated QoS requirements. Once the "set" of best suited components have been found, the network must select one of them to be instantiated attending to characteristics such as location, cost, migration, etc. Once selected, the network can decide either to instantiate the component in its original node or to fetch the component to be locally installed, instantiated and run. For example, a component decoding a MPEG video stream would work much faster if it is installed locally.

This network behavior is implemented by the ***Distributed Registry***. It stores information covering the resources available in the network as a whole, and is responsible of managing these. Whenever the network must decide where to run components or if some component instances must migrate, it uses the Distributed Registry infrastructure to perform these decisions.

Component Registries, Resource Managers and the Network Cohesion interface of each node support the Distributed Registry behavior. Component Registries collaborate to resolve distributed component queries and reflect the internal Component Repository of each node. Populating the node's Component Repository makes the Distributed Registry aware of the change. This simplifies the application management greatly [13]. Similarly, Resource Managers supply node's resource utilization to help the Distributed Registry decisions.

The Distributed Registry activities include:

- ***Logical Network Management***: encompasses the protocol used
    1. to maintain the "logical" connection between them: which nodes are available, message routing, ping/reply handshaking, etc., and
    2. to maintain updated information about the available resources in the network. This information includes the meta-data given by the Reflection Architecture in each node.
- Support for ***Distributed Queries***: As the Distributed Registry stores information regarding the whole network, it is also in charge of resolving distributed component queries.
- ***Network Resource Monitoring*** and component instance migration and replication to achieve load balancing and fault tolerance. Resource Managers help the Distributed Registry The Resource Manager interface also supports the ***Meta-Resource Managers (MRMs)*** operations. Meta-Resource Managers, instead of managing one machine resources, maintain an updated view of a set of node's Resource Managers. This allows a hierarchical treatment of network resources, simplifying the network management.

Obtaining the network management required by the Distributed Registry behavior in distributed environments is never an easy task. To be realistic in a truly distributed, scalable system, the protocol must support spurious node failures and node disconnections (and re-connections) gracefully. Also, the network must integrate seamlessly the whole set of nodes present in a given system. As the number of nodes can become arbitrarily large, the need for efficient protocols arise. The logical network extension can also become potentially large, raising the need for hierarchical protocols [13]. Concretely, the guidelines we have identified for the network management protocol include:

- **Hierarchical protocol**: The protocol must allow logical grouping and incremental resource lookup. If current requirements cannot be met with current level resources, the protocol must request higher hierarchy level requests. This reduces network load and exploits locality. The protocol must also carry group formation deciding the nodes that are going to implement the Meta-Resource Manager interface [2]. Each MRM manages a group of nodes or a group of other MRMs, maintaining this hierarchical structure and behavior.
- **"Soft" network consistency**: Instead of maintaining a "strong" network consistency in which MRMs have perfect knowledge of the set of hosts they manage, MRMs have an approximate view of the present resources. For instance, the nodes can send to the MRM periodical updates of their resource availability which also serve as a "keep-alive" mechanism. The MRM can suppose a node of the group has been down after some time-out, and must support either node disconnections and re-connections gracefully. This soft consistency protocol leads to lower bandwidth utilization and better scalability. Also, predictive and adaptive techniques can be used to predict the resource availability, thus reducing even more the bandwidth requirements.
- **Peer-replicated protocol**: To enhance fault-tolerance, the protocol must allow replicated peer MRMs per group. The number of these replicas must be decided by the protocol depending on FT requirements. Again, the protocol must adapt by creating new replicas as needed and catching replica failures.

As stated in §5, we are testing implementations of these protocols using techniques such as multicast, event and notification channels [23], asynchronous messaging [20] and object group service [5].

2.4.4. Applications

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 (how many instances and the name of each, of which components, how are them interconnected), and (2) they are created by users with the help of visual building tools.

With the given definition, applications can be considered as ***bootstrap*** components: when applications start running, they expose their explicit dependencies, requiring instances of other components and connecting them following the user stated pattern for that particular application. This is similar to what in CCM is called an ***assembly***. Conversely, in CORBA–$\mathcal{LC}$ the matching between component required instances and network-running instances is performed at run-time: the exact node in which every instance is going to be run is decided when the application requests it, and this decision may change to reflect changes in the load of either the nodes or the network.

Thus, the deployment of the application, instead of being fixed at deployment-design time, is intelligently performed at run-time. This flexibility allows the implementation of intelligent run-time scheduling, migration and load balancing schemes. The difference between fixed and run-time deployment is similar to the difference between static and dynamic linking of Operating Systems libraries, but augmented to the distributed, heterogeneous case.

3. Application domains

The CORBA–$\mathcal{LC}$ model represents a very convenient infrastructure for developing applications in a wide range of domains. It can be seen as a general purpose infrastructure. However, we are specially interested in dealing with CSCW and Grid Computing.

3.1. Computer Supported Cooperative Work (CSCW)

Collaborative work applications allow a group of users to share and manipulate a set of data (usually multi-media) in a synchronous or asynchronous way regardless of user location [34]. We are interested in the development and deployment of ***synchronous*** CSCW applications, including videoconferencing, shared whiteboard and workspaces, workflow and co-authoring systems. CORBA–$\mathcal{LC}$ represents an optimal environment for various reasons:

- It offers a peer distributed model, which matches the inherently peer distributed nature of these applications.
- GUI components can be considered within the modular design of the application, thus allowing the replacement of the presentation layer to suit additional user or application needs.
- It allows bandwidth-limited multimedia components (such as video stream decoding) to be migrated and installed locally to minimize network load.
- It allows tiny devices such as Personal Digital Assistants (PDAs) to be used as normal nodes with limited capabilities: they can use all components remotely.

Figure 2 depicts the relationships between a CSCW application and other components, including GUI components. The latter can be either local or remote, and use the local *Display* component providing painting functions. Each GUI component is in charge of a portion of the window, and applications can change how the data is shown by replacing the GUI components with others at run-time. Note that all components required by the application can be remote, thus allowing the use of thin clients such as PDAs.

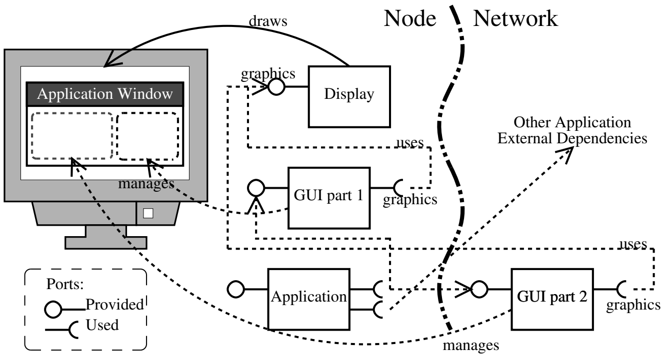

Figure 2. CSCW application model.

3.2. Grid Computing

Our view of Grid Computation targets scalable and intelligent resource and CPU usage within a distributed system, using techniques such as *IDLE computation* [8] and *volunteer computing* [26]. These techniques fit seamlessly within the CORBA–$\mathcal{LC}$ model to suit Grid Computation needs.

Other component-based alternatives such as the Common Component Architecture (CCA) [1] and Ligature [12] have appeared in the High-Performance Computing (HPC) community. These models address the needs of scientific computing, introducing components kinds which reflect the special characteristics of the field (for example, components whose instances must be split and distributed into the network to perform a highly-parallel task). While we find this approach very interesting, those models usually become only a minimum wrapper [14] for reusing legacy scientific code and do not offer a complete component model.

FOCALE [3] offers a component model for grid computation. It uses CORBA and Java (although it supports legacy applications). It provides a system view at different levels: federation, server, factories, instances and connections.

Notable developments in the Metacomputing and Grid Computing fields include Globus [6] and Legion [10]. They are systems which offer services for applications to access to the computational grid. However, they are huge systems, difficult to manage and configure, somewhat failing in its primary intentions. Moreover, they do not address very well the interoperability and code reuse through component technology.

Recent interest has been shown by the OMG regarding parallel applications [21] and aggregated computing [17]. However, the OMG has neither agreed nor released any specification on these topics.

4. Related Work

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

Java Beans [30] is a framework for implementing Java-based desktop applications. It is limited to both Java and the client side of the application. In contrast to Java Beans, CORBA–$\mathcal{LC}$ is not limited to Java and allows components to be distributed among different hosts, still allowing seamless integration of local GUI components.

Microsoft's Component Object Model (COM) [16] offers a component model in which all desktop applications are integrated. In our opinion, its main disadvantages are that, from a practical point of view, (1) it does not integrate very well the distributed case (DCOM) and (2) its support is rather limited to the Windows Operating System. Moreover, COM components do not expose their requirements (other required components) [25, 13]. CORBA–$\mathcal{LC}$ inherits from CORBA its Operating System, programming language and location transparency, thus effectively adapting to heterogeneous environments. Moreover, it is designed from the beginning to automatically exploit the computing power and components installed in all hosts participating using its Reflection Architecture.

In the server side, SUN's EJB [31] and the new Object Management Group's CORBA Component Model (CCM) [18, 28] offer a server programming framework in which server components can be installed, instantiated and run. Those models are fairly similar. In fact, CCM "*basic*" level makes both models totally compatible. EJB is a Java-only system, while CCM continues the CORBA heterogeneous philosophy. Both models are designed towards supporting enterprise applications, thus offering a container architecture with convenient support for transactions, persistence, security, etc. [27] They also offer the notion of components as binary units which can be installed and executed (following a fixed assembly) in Components Servers (or Application Servers in the case of EJB).

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. It also allows adding new components and modifying component instances properties and connections at run-time and reflecting those changes to visual building tools. 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: it does not incur in the overhead of offering services such as transactions and persistence. This overhead and complexity is one of the main reasons why complete mature CCM implementations are not expected in 2–3 years. In fact, the CCM specification still has open issues such as the *Component Implementation Definition Language* (CIDL) mapping, is not finished (as of April, 2001), and only one preliminary implementation exists: OpenCCM [9].

In general, component models have been designed to be either client-side ***or*** server-side. This forces programmers to follow different models for programming the different layers of applications. CORBA–$\mathcal{LC}$ offers a more *peer* approach in which applications can utilize all the computing power available, including the more and more powerful user workstations and high-end servers. Application components can be developed using a single component model and spread into the network. They will be intelligently migrated into the required hosts. Thus, a *homogeneous* component model can be used to develop all the tiers (GUI, application logic) of distributed multi-tiered applications.

In [13], a dynamic configuration management system is described. This work provides us with valuable ideas for our research. However, it is centered in the process of automatic component configuration and does not offer a complete component model.

5. Conclusions, Status and Future Work

In this article we have described the requirements identified in the design and preliminary implementation of the CORBA–$\mathcal{LC}$ Component Model. Also, we have stated the validity of the design to target the CSCW and Grid Computing domains. Current CORBA–$\mathcal{LC}$ implementation allows building components with the stated external characteristics and packaging. However, the implementation is still incomplete, so we have some future work to do:

- Explore strategies to maintain the described Reflection Architecture and the network-awareness of both nodes and the Distributed Registry [13], also introducing fault-tolerance [22], migration, replication and load-balancing techniques [24].
- Implement visual building tools allowing users to build applications based on all available network components.
- Further identify CSCW and Grid-based application needs [17, 12] enhancing CORBA–$\mathcal{LC}$ to better support them. We think our research can help OMG efforts in this direction, as no specification currently exists.
- Study the integration of this model with current and future CCM implementations [9].

Finally, we plan to continue enhancing CORBA–$\mathcal{LC}$ as a general computing platform, to offer programmers both the advantages of the Component-Based Development and Distributed Computing.

References

[1] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A Component Based Services Architecture for Building Distributed Applications. In *Proceedings of the High Performance Distributed Computing Conference*, 2000.

[2] F. Cristian and F. Schmuck. Agreeing on Processor Group Membership in Asynchronous Distributed Systems. Technical Report CSE95-428, Department of Computer Science & Engineering, University of California, San Diego, California, 1995.

[3] G. S. di Apollonia, C. Gransart, and J.-M. Geib. FOCALE: Towards a Grid View of Large-Scale Computation Components. In *Grid'2000 Workshop, 7th Int. Conf. on High Performance Computing*, Bangalore, India, Dec. 2000.

[4] J. Fabry. Distribution as a set of Cooperating Aspects. In *ECOOP'2000 Workshop on Distributed Objects Programming Paradigms*, June 2000.

[5] P. Felber. *The CORBA Object Group Service. A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, EPFL, 1998.

[6] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishing, 1999.

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

[8] D. Gelernter and D. Kaminsky. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. In *Sixth ACM International Conference on Supercomputing*, pages 417–427, July 1992.

[9] GOAL group, Laboratoire d'Informatique Fondamentalle de Lille. *OpenCCM, v0.2*, 2001. http://corbaweb.lifl.fr/OpenCCM/, visited April, 2001.

[10] A. S. Grimshaw and W. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

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

[12] K. Keahey, P. Beckman, and J. Ahrens. Ligature: Component Architecture for High-Performance Applications. *The International Journal of High Performance Computing Applications*, 14(4):347–356, Winter 2000.

[13] F. Kon, T. Yamane, C. Hess, R. Campbell, and M. Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, Texas, February 2001.

[14] M. Li, O. F. Rana, M. S. Shields, and D. W. Salker. A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components. In *Supercomputing'2000 Conference*, Dallas, TX, November 2000.

[15] 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)*, Reims, France, May 2000.

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

[17] Object Management Group. *Aggregated Computing in CORBA RFI*, 1999. OMG Document orbos/99-01-04.

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

[19] Object Management Group. *CORBA: Common Object Request Broker Architecture Specification, revision 2.4.1*, 2000. OMG Document formal/00-11-03.

[20] Object Management Group. *CORBA Messaging*, 2000. OMG Document ptc/00-02-05.

[21] Object Management Group. *Data Parallel Application support for CORBA RFP*, 2000. OMG Document orbos/00-03-17.

[22] Object Management Group. *Fault Tolerant CORBA Specification*, 1.0 edition, 2000. OMG Document ptc/2000-04-04.

[23] Object Management Group. *Notification Service*, 1.0 edition, 2000. OMG Document formal/2000-06-20.

[24] O. Othman, C. O'Ryan, and D. Schmidt. The Design and Performance of an Adaptative CORBA Load Balancing Service. *Distributed Systems Engineering Journal*, 2001.

[25] 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.

[26] L. F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. In *2nd International Conference on World-Wide Computing and its Applications (WWCA'98)*, March 1998.

[27] D. Sevilla. CORBA & Components. Technical Report TR-12/2000, University of Extremadura, Spain, 2000.

[28] D. Sevilla. *The CORBA & CORBA Component Model (CCM) Page*, 2001. http://www.ditec.um.es/˜dsevilla/ccm/, visited April, 2001.

[29] D. Sevilla, J. M. García, and A. Gómez. CORBA Lightweight Components: A Model for Distributed Component-based Heterogeneous Computation. Technical Report UM-DITEC-2001-04, Dept. of Computer Engineering, University of Murcia, Spain, February 2001.

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

[31] SUN Microsystems. *Enterprise Java Beans specification*, 1.1 edition, December 1999. http://java.sun.com/products/ejb/index.html.

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

[33] C. Szyperski. Beyond Objects. *Software Development Online*, 2001. http://www.sdmagazine.com/feature/uml/beyondobjects/.

[34] G. H. ter Hofte. *Working Apart Toguether. Foundation for Component Groupware*. PhD thesis, Telematica Institut, The Netherlands, 1998.

[35] WWW Consortium. *The Open Software Description Format (OSD)*, August 1997. http://www.w3.org/TR/NOTE-OSD.html.