# An Architectural Perspective of the static invocation in CORBA

*Carlos E. Cuesta, Pablo de la Fuente and Manuel Barrio-Solórzano*
*Departamento de Informática, Universidad de Valladolid, Spain*
*Camino del Cementerio s/n, 47011 Valladolid, Spain.*
Tel: *+34 983 42.36.70*. Fax: *+34 983 42.36.71*.
E-mail: {*cecuesta,pfuente,mbarrio*}*@infor.uva.es.*

**Abstract**

Software architecture is becoming essential for the development and maintenance of distributed systems. It provides the concept of *style*, which refers to systems with similar features. CORBA systems share a common framework, so they can be described as conforming such a style. This paper gives a first step towards its formalization, by giving a comparison of the concepts of component and description language, and studying in detail a static invocation system, expressed in $\delta$arwin. Finally, it discuss the need of more dynamic structures than in current Software Architecture.

**Keywords**

Software Architecture, Component, Architectural Style, ADL, CORBA, Static Invocation, $\delta$arwin.

## 1 INTRODUCTION

Software systems are shifting from monolithic to complex, distributed schemas. Structure is becoming a critical aspect in development, reuse and maintenance. New abstractions are required to deal with this, and perhaps the most promising is *Software Architecture*, which has been defined as *"the structure of the* components *of a program/system, their* interrelationships, *and principles and* guidelines *governing their* design *and* evolution *through time"* [4].

CORBA standard [8] gives us a frame architecture which eases the construction of new distributed systems. They acquire common features, sharing an infrastructure which should be formalized in terms of Software Architecture.

This paper is just a first step in this direction: CORBA systems are viewed as concrete configurations of a common architectural *style*, and a conceptual comparison is provided.

To get a practical perspective, a sample system, based on *static invocation*, is architecturally described in $\delta$arwin. This ADL was chosen because its object-oriented focus and dynamic capabilities make it specially adequate for our system.

In the following, we give an introduction to Software Architecture; then we consider the perspective of CORBA as a style, and discuss the similarities in the concepts of component and description language. Finally, the review of the static invocation system shows the need for more dynamic constructions in existing ADLs.

## 2 INTRODUCTION TO SOFTWARE ARCHITECTURE

In the introduction we quoted one of the many existing definitions of software architecture. In short, it deals with the formalization of the global structure of a system, emphasizing the study of the interaction between its basic elements, named *components*. Any system can be seen as a *composition* of such components; the layout will be its *architecture*.

The main feature of a component is that it has an *interface* which expresses its relationship to the rest of the architecture. The interface is usually segmented in independent blocks called *ports* (*portals* in $\delta$arwin). If specific communication components (*connectors*) are defined, their interface will be segmented in *roles*.

An architecture is defined by the creation of different *instances* of each *type* of component, and the joining of them all in a structure through specific *bindings*. This is called a *configuration*, and is usually considered within a hierarchy.

When we want to study the generic patterns that define a family of systems instead of describing a particular configuration, we speak of *architectural style*. The definition of styles is fundamental from the point of view of reuse, and essential for Software Architecture development.

The *Architecture Description Languages* (ADLs) were proposed as a way to express these concepts in an organized manner. In spite of having this common context, these languages have very different visions [2]. Among the most interesting are Aesop, UniCon, C2, Rapide, Wright and $\delta$arwin. In general, which one we use depends on the type of system we are dealing with.

$\delta$arwin originates in real experience with distributed systems, but it has also been formalized in $\pi$-calculus [5, 6]. So it maintains both a high conceptual level and practical sense. It has been chosen as the ADL for this paper, due to its special suitability for CORBA, and to our particular case study.

## 3 AN ARCHITECTURAL VISION

In the following, we review CORBA in the light of Software Architecture concepts, showing their similarities.

### 3.1 CORBA as an Architectural Style

Although CORBA is often referred as an architecture, we must point out that it is an *architectural style*, in our definition. It presents a general distributed schema, the basis for building many particular configurations.

This implies that when attempting to systematize a *"CORBA architecture"* we are not referring to a single system, but to all the variations that could appear in a general interaction pattern. Then it is this pattern –CORBA– what must be modelled, to serve as a foundation for later developments.

This isn't trivial, because every CORBA configuration is variable. Simple systems can evolve over time into complex structures. Our style should be flexible enough to allow such changes.

Distributed systems tend towards CORBA or similar frameworks. To discover the implications, a complete description of the aforementioned style is indispensable. Software architecture will provide the means to systematize them. This formalization will make possible to verify the systems which conform to the style. It will also enforce the reuse of components, by providing tested configurations and a common framework. In short, it's essential for the future development of distributed systems.

## 3.2 The concept of component

Architectural and implementation components are close but not identical concepts. We should note its coincidences and divergences.

An architectural component is an abstract autonomous element with a specific interface, which dissociates its interactional and functional aspects. It is an element at the design level.

Meanwhile, an implementation component is kind of a "distributed object", even if not object-oriented. What identifies a CORBA component, actually, is its interface definition in IDL. This provides a fixed interaction pattern, together with language independence.

There are undeniable similarities. First, by the separation between functionality and interaction, even using different languages. Second, by encapsulation; the boundaries of a component are outlined by their interfaces. Finally, those interfaces are segmented: implementation public methods (services) correspond to architectural ports. This is most easily seen on CORBA than in any other distributed environment.

Apparently, CORBA is an ideal medium to implement the abstractions of Software Architecture. However, the equivalence is partial, and the abstraction level is quite different. Specifically, it has a single kind of interaction, opposite to all possible architectural connectors. CORBA is bound to procedure call semantics, thus it's less expressive than any ADL.

## 3.3 The Description Language

As suggested above, CORBA is particularly interesting ought to the IDL. This Interface Description Language is what allows the separation between interface and implementation, providing language independence and raising the abstraction level.

Current Software Architecture already deals with components with diverse granularity: from basic elements up to complete subsystems. Any configuration can be seen as a higher-order component, in a hierarchical structure. The opposite is also true: a component is a lower-order configuration.

Future Software Architecture would comprise the whole design process. An initial architecture will be refined into more and more complex components, until implementation-level detail is reached. Then each component is implemented, reusing the existing ones and creating the rest from skeletons generated by interface descriptions. This guarantees the coherence in the interactions, and allows implementors to care just about pure funcionality.

CORBA IDL foreshadows these code-generation issues, and is fairly consistent with the rest of the exposed ideas. Given all that, it seems quite near to be a “real” ADL. But the resemblance it’s not close enough. IDL is limited to use procedure calls, and it doesn’t have any explicit semantics. Furthermore, bindings, instances, and concrete configurations are given by the dynamics of the system, and can’t be expressed within the language, then losing the architectural claim. Finally, IDL is too influenced by implementation aspects, such as the use of strong typing and multiple inheritance, whose presence in Software Architecture is rather debatable.

## 4 A CORBA ARCHITECTURE IN $\delta$arwin

In order to have a practical perspective of the meaning of this conceptual correspondences, we review a particular CORBA system in terms of a concrete ADL.

Our case study consciously leaves out most CORBA dynamic features, to avoid the intersection of different concerns, specially those regarding evolution. It is both a minimal example and the description of CORBA’s most basic interaction: the *static invocation*. Even this architecture shows a high degree of dynamism. Few ADLs are capable of reflecting these facts; that’s why $\delta$arwin [3] has been chosen, despite its other drawbacks.

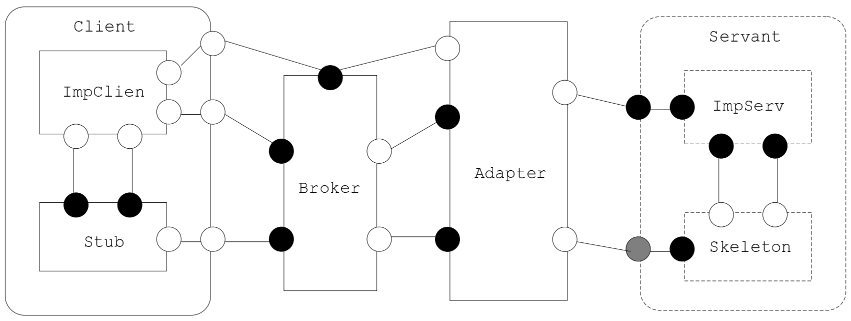
In the following, a basic knowledge of CORBA mechanisms will be assumed; further details can be easily found in related bibliography [8, 9].

### 4.1 The Minimal Static Invocation

Initially, we distinguish four component types: Client, Broker (ORB core), Adapter and Servant (Server). We consider just one instance of each type. The components interact in a standard static invocation, following the schema shown in Figure 1.

The communication paths are divided in two:

- **Initialization.** Client and Adapter present themselves to the Broker. When the Client tries to obtain its server, a new Servant is dynamically instantiated and registered. This happens only once.
- **Invocation(s).** Client tries to invoke a service. A message is sent through the



```
component Static {
inst
  C: Client;
  B: Broker;
  A: Adapter;
  S: dyn Servant;
bind
  C.init -- B.init;
  A.init -- B.init;
  C.s_id -- B.search_id;
  A.s_id -- B.s_id;
  A.call -- S.service;
  C.call -- B.callee;
  B.caller -- A.callee;
  A.serv_id -- S.iden;
  }
```

```
component Broker {
  require init;
  provide search_id;
  provide callee;
  require s_id;
  require caller;
  }
```

```
component Adapter {
  require init;
  provide s_id;
  provide callee;
  require serv_id;
  require call;
  }
```

```
component Client {
  require init;
  require s_id;
  require call;
  }
```

```
component Servant {
  provide iden;
  provide service;
  }
```

```
component Client  {
  require init;
  require s_id;
  require call;
inst
  IC: ImpClien;
  CC: Stub;
bind
  IC.servic1 -- CC.servic1;
  IC.servic2 -- CC.servic2;
  IC.init -- init;
  IC.s_id -- s_id;
  CC.call -- call;
  }

component ImpClien {
  require init;
  require s_id;
  require servic1;
  require servic2;
  }

component Stub {
  provide servic1;
  require servic2;
  require call;
  }
```

```
component Servant {
  provide s_id;
  provide service;
inst
  IS: ImpServ;
  E:  Skeleton;
bind
  IS.servic1 -- E.servic1;
  IS.servic2 -- E.servic2;
  E.service -- service;
  IS.s_id -- s_id;
  }

component ImpServ {
  provide s_id;
  provide servic1;
  provide servic2;
  }

component Skeleton {
  require servic1;
  require servic2;
  provide service;
  }
```

**Figure 1** Static Invocation Model in δarwin: ver. 1 & 2

Broker and Adapter, and the task is carried out by the Servant. The result is then sent back.

The Broker is responsible to find the right Adapter, which acts as the Servant's mediator. The paths are obvious, given that *provide/require* bindings are asymmetrical, but bidirectional.

Two particular facts are architecturally significant:

- *Servant's Creation.* The Adapter must instance a new Servant when it is requested; the need for dynamic constructors arises from here. Fortunately, $\delta$arwin provides an adequate abstraction in the form of *lazy instantiation*. Required components (Servant) are defined, but not created until they are accessed, as the Figure specifies (dyn S).
- *Marshalling.* Each Client's remote call is transformed into a message, and decoded in the Servant; the same happens with the corresponding answer. This is a defining feature of CORBA, an it's carried on by two IDL-generated communication elements, namely the Client's **Stub** and the Servant's **Skeleton**.

We should then refine the specification to express this point. The abstraction level is lowered, and Client and Servant are seen as the aggregation of an implementation component (ImpClien, ImpServ), which does the real job, and a *connector*, a concept that $\delta$arwin has to express with a normal component.

This second version is the extension also in Figure 1. The overall architecture is still the same: all the changes are within the limits of pre-existing interfaces. It's important to note that *lazy instantiation* covers now both ImpServ and its i Skeleton, which are created simultaneously.

## 4.2 Extending the Case Study

The exposed schema is a working model, and it uses several implicit assumptions, which become significant in further development. We explore some of them in the following.

- *Adapter Component.* The Adapter specification should be re-elaborated. Its current form shows some difficulties. For instance, the portal call is of the *require* type, allowing it to bind to a single Servant. Each new Servant requires a new portal, so our Adapter is too limited.
  Besides, CORBA has *four* different Adapter models, and all of them should be considered. Some of them, however, require even more dynamism than the one exposed.
- *Lazy Instantiation Model.* Currently, the Servant is lazy-instanced by an identification request. Then, $\delta$arwin's *direct instantiation* mechanism could seem more adequate, with a portal create acting as an object *constructor*. Nevertheless, this

would invert all the portals considered, as they only can be of the *require* type. We end up with a "server" which requires services, instead of providing them; this paradox must be rejected.
Lazy instantiation doesn't have this problems, though the activating portal (iden, pictured in grey) must be a *provide*, opposite to the usual case. However, this is even more consistent with the semantics of our system.

- *Cardinality.* We should consider the addition of new instances of the given components. Clients don't pose any problems: the portals in the Broker allow such situation. Multiple Servants would cause the conflicts already exposed with the Adapter. Multiple Adapters would solve this problems, at the cost of creating new ones, now with the Broker itself.

To solve this problems we need a new Adapter model. However, this would require a dynamism that current ADLs doesn't have yet.

## 5 CONCLUSIONS

Architectural modelling of CORBA is not simple. We are dealing here with an evolutionary framework, which uses widely a dynamism that Software Architecture has seldom considered until recently.

$\delta$arwin has been a pioneer in this interest [6], and it manages to handle the system somehow, thanks to the asymmetry of its bindings and its (lazy) instantiation constructs. Even CORBA's object-reference passing is better resembled by mobile names in its $\pi$-calculus formalization [5] than any manipulation of CSP labels [1].

However, $\delta$arwin semantics are not flexible enough; it would be interesting to detail the protocols of the portals. If we change from *service* to *behavioural view* [7], which makes it possible, the model should be reviewed. Anyway, $\delta$arwin still lacks the concept of *connector* and, specially, *style*. This is important, given that our long-term goal is exactly this (CORBA *style*).

Further development should then change the ADL; but none of them is completely suitable. Such models like dynamic invocation or evolutionary nature of ORBs can't be expressed in current languages. This is not an exception: to reach the next stage in its development, Software Architecture must tackle the study of dinamism.

## REFERENCES

[1] ALLEN, R., DOUENCE, R., AND GARLAN, D. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of 1998 Conference on Fundamental Approaches to Software Enginnering* (Mar. 1998).

[2] BARRIO, M., AND DE LA FUENTE, P. Software Architecture: Object vs. Process Approach. In *Proceedings of SCCC'97* (Nov. 1997).

[3] DULAY, N. The Darwin Language: Version 3c. Imperial College, Mar. 1997.

[4] GARLAN, D., AND PERRY, D. E. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering 21*, 4 (Apr. 1995).

[5] MAGEE, J., DULAY, N., EISENBACH, S., AND KRAMER, J. Specifying Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference* (Sept. 1995).

[6] MAGEE, J., AND KRAMER, J. Dynamic Structure in Software Architectures. *Software Engineering Notes 21*, 6 (Nov. 1996), 3–14.

[7] MAGEE, J., KRAMER, J., AND GIANNAKOPOULOU, D. Analysing the Behaviour of Distributed Software Architectures: a Case Study. In *5th IEEE Workshop on Future Trends of Distributed Computing Systems* (Oct. 1997).

[8] OMG. *The Common Object Request Broker: Architecture and Specification. Revision 2.2*. OMG, July 1998.

[9] ORFALI, R., HARKEY, D., AND EDWARDS, J. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996.