CORBA & DSA: Divorce or Marriage?

Laurent PAUTET, Thomas QUINOT, and Samuel TARDIEU

{pautet,quinot,tardieu}@enst.fr
ENST
Paris, France

Abstract

This paper presents a comparison between CORBA and the Ada 95 Distributed Systems Annex. We also focus on the latest developments made by the ENST research team to GLADE that are related to CORBA services.[1]

1 Introduction

Before comparing two models for distributing heterogeneous applications, one must realize that they have very different goals. The Object Management Group (OMG) was formed to promote standards for the development of distributed heterogeneous applications. The Common Object Broker Request Architecture (CORBA) is the key component of its Object Management Architecture (OMA). CORBA [9] is an architecture for interoperable distributed object systems.

The Distributed System Annex (DSA) of Ada 95 pursues a different objective. It provides a model for programming distributed systems within the disciplined semantics of a language that supports type-safe object-oriented and real-time programming. From our experience, DSA has very powerful features compared to CORBA and offers an interesting balance between abstraction and performance. CORBA specifies a flexible architecture based on interoperable and reusable components. Our general objective is to propose the interesting features of CORBA to DSA users.

[1] The current release of GLADE, the implementation of Annex E of the Ada Reference Manual for the GNAT compiler, has been developed by the ENST team and is maintained by ACT Europe. See `http://www.act-europe.fr/`.

1.1 Programming models for distributed systems

Using OS network services In [11] we present several programming techniques for developing distributed applications. These applications have traditionally been developed using network programming interfaces such as TCP or UDP sockets. Programmers explicitly have to perform calls to operating system services, a task that can be tedious and error-prone. This includes initializing socket connection and determining peer location, marshalling and unmarshalling data structures, sending and receiving messages, debugging and testing several programs at the same time, and porting them on several platforms to account for subtle differences between network interfaces.

Of course, this code can be encapsulated in wrappers to reduce its complexity but it is clear that most of it could be automatically generated [12]. Message passing diverts developer's attention from the application domain. The query and reply scenario is a classical scheme in distributed applications; using message passing in such a situation could be compared to using a "goto" mechanism in a non-distributed application. This is known to cause significant problems with respect to modern programming languages. A more robust design would be to use a structured approach based on procedure call.

Using a middleware environment A middleware environment is intended to provide high level abstractions in order to ease development of user applications. Environments like CORBA or Distributed Computing Environment (DCE) offer an approach to develop client/server applications using the Remote Procedure Call model (RPC). The RPC model [1] is inspired from the query and reply scheme. Compared to a regular procedure call, arguments are pushed into a stream along with some data specifying which remote procedure is to be used. The stream is then transmitted over the network to the server. The server decodes the stream, does the regular subprogram call, then put the output parameters into another stream along with the exception (if any) raised by the subprogram, and sends this stream back to the caller. The caller decodes the stream and raises the exception if needed.

CORBA provides the same enhancements to the remote procedure model that object languages provide to classical procedural languages. This includes encapsulation, inheritance, type checking, and exceptions. These features are offered through an Interface Definition Language (IDL).

The middleware communication framework provides all the machinery to perform, somewhat transparently, remote procedure calls or remote object method invocations. For instance, each CORBA interface communicates through an Object Request Broker (ORB). A communication subsystem such as an ORB is intended to allow applications to use objects without being aware of their underlying message passing implementation. But the user may also require a large number of complex services to develop the distributed application. Some of them are definitively needed like a location service that allows clients to reference remote services via higher level names instead of a traditional scheme for addressing remote services involving Internet host addresses and communication port numbers. Other services provide domain independent interfaces that are frequently used by distributed applications like naming services.

Using a distributed language Rather than defining a new language like an IDL, an alternative idea is to extend a programming language in order to provide distributed features. The distributed object paradigm provides a more object-oriented approach to programming distributed systems. The notion of a distributed object is an extension to the abstract data type that permits the services provided in the type interface to be called independently of where the actual service is executed. When combined with object-oriented features such as inheritance and polymorphism, distributed objects promote a more dynamic and structured computational environment for distributed applications.

Ada 95 includes a Distributed Systems Annex (DSA) which defines several extensions allowing a user to write a distributed system entirely in Ada, using packages as the definition of remote procedure call or remote method call on distributed objects [3]. The distributed systems models of Ada 95, Java/RMI [7], and Modula-3 [2] are all very close, and all replace IDL with a subset of the language. The language supports both remote procedure calls and remote object method invocations transparently.

A program written in such a language is supposed to communicate with a program written in the same language, but this restriction also yields useful consequences. The language can provide more powerful features because it is not constrained by the smallest common subset of features available in all host languages. In Ada 95, the user defines a specification of remote services and implements them exactly as he would for ordinary, non-distributed services. The Ada 95 environment compiles them to produce a stub file and a skeleton file that automatically includes calls to the actual service *body*. Creating objects, obtaining or registering object references or adapting the object skeleton to the user object implementation are transparent because the language environment has a full control on the development process.

2 DSA *vs.* CORBA comparison

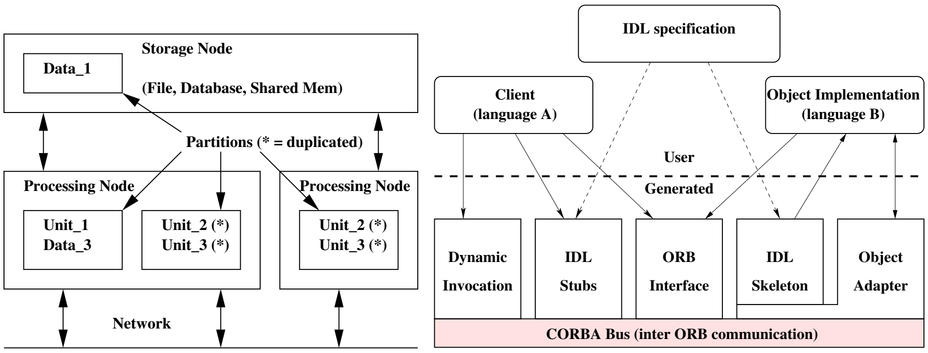
2.1 Overview of DSA

The Ada Distributed Systems Annex provides a solution for programming distributed systems. An Ada application can be partitioned for execution across a network of computers so that typed objects may be referenced through remote subprogram calls. The remotely-called subprograms declared in a library unit categorized as remote call interface (RCI) or remote types (RT) may be either statically or dynamically bound, thereby allowing applications to use one of the following classical paradigms:

Remote subprograms: for the programmer, a remote subprogram call is similar to a regular subprogram call. Run-time binding using access-to-subprogram types can also be used with remote subprograms.

Distributed objects: particular access types can be defined, which designate remote objects. When a primitive dispatching operation is invoked on an object designated by a remote access, a remote call is performed transparently on the partition on which the object was created.

Shared objects: data can be shared between active partitions, providing a repository similar to a shared memory, a shared file system or a database. Entryless protected objects allow safe access and update on shared objects. This feature is orthogonal to the notion of distributed objects, which are only accessed through exported services.



(a) Overview of DSA (b) Overview of CORBA

Fig. 1. Comparing the two architectures

An Ada 95 distributed application is a set of partition executed concurrently on one or more machines; each partition is constituted of one or more compilation units which together constitute an executable binary.

2.2 Overview of CORBA

CORBA is an industry-sponsored effort to standardize the distributed object paradigm via the CORBA Interface Definition Language (IDL). The use of IDL makes CORBA more self-describing than any other client/server middleware. [10] describes the main features of CORBA, which are Interface Definition Language, Language Mappings, Stubs, Skeletons and Object Adapters, ORB, Interface Repository, Dynamic Invocation, ORB protocols and CORBA services.

The IDL specifies modules, constants, types and interfaces. An object interface defines the operations, exceptions and public attributes a client can invoke or access. CORBA offers a model based solely on distributed objects. In some respects, it can be compared to Java, as this language only provides an object-oriented programming model, and ignores the classical structured programming model.

An IDL translator generates client stubs and server skeletons in a host language (e.g., `C++`, C, Java, Ada 95) [9]; a language mapping specifies how IDL entities are implemented in the host language. Depending on the features available in the host language, the mapping can be more or less straightforward. When an IDL feature is not defined in the host language, the mapping provides a standardized but complex way of simulating the missing feature. Although the user works with the generated code, a good understanding of the language mapping is often necessary.

When the host language does not provide object-oriented features, the user has to deal with a complex simulation of those functions. A `C++` programmer has to follow several rules related to parameters passed by reference. Defining whether the callee or the caller is responsible for parameter memory allocation can be considered as `C++`

programming conventions. The most difficult parts of the Ada mapping, that an Ada programmer should avoid when possible, are multiple inheritance and forward declarations.

The IDL translator produces several host language source files depending on the language mapping: client files called *stubs* and server files called *skeletons*. These files are specific to a vendor and product, as they make calls to a proprietary communication subsystem, but their structure and interface are supposed to follow a standard canvas. The client stubs convert user queries into requests to the ORB, which transmits these requests through an object adapter to the server skeleton (figure 1(b)).

2.3 Interface Definition Language

In DSA, the IDL is a subset of Ada 95. The user identifies interface packages at compile time. Some library-level packages are categorized using pragmas:

Remote Call Interface (RCI): Library units categorized with this pragma can declare subprograms to be called and executed remotely. This RPC operation is a statically bound operation. In these units, clients and servers do not share their memory space.

Dynamically bound calls are integrated with Ada capabilities to dereference subprograms (remote access to subprogram — RAS) and to dispatch on class-wide operands (remote access on class wide types — RACW). These remote access types can be declared in an RCI package.

A remote access type can be seen as a fat pointer — a structure with a remote address and a local address. The remote address can describe the host on which the entity has been created; the local address describes the service in the remote address space.

Remote Types (RT): Unlike RCI units, library units categorized with this pragma can define distributed objects and remote methods on them. They can also define the remote access types described above. A subprogram defined in a RT unit is not a remote subprogram. Unlike RCI units, a RT unit are not to be placed on only one partitions, they act like regular units instead.

Shared Passive (SP): the entities declared in such library units are to be mapped on a shared address space (file, memory, or database). When two partitions use such a library unit, they can communicate by reading or writing a common variable. This corresponds to the shared variables paradigm. Entry-less protected objects declared in these units provide atomic access to shared data, akin to in a transaction.

In RT or RCI units, variables are forbidden and non-remote access types are allowed only as long as their marshalling subprograms are provided. Any exception raised in a remote method or subprogram call is propagated to the caller.

An additional pragma All_Calls_Remote in a RCI unit can force a remote procedure call to be routed through the communication subsystem even for a local call. This allows debugging of an application in a non-distributed situation that is close to the distributed case.

A pragma Asynchronous allows statically and dynamically bound remote calls to be executed asynchronously. An asynchronous procedure doesn't wait for the completion of the remote call and lets the caller continue its execution path. The procedure must have only *in* or *access* parameters, and any exception raised during the execution of the remote procedure is lost.

All such categorized units can be generic. Instances of these generic packages can be either categorized or not. In the latter case, the unit loses its categorization property.

Each categorization pragma has very specific dependancy rules. As a general rule, RCI $\succ$ RT $\succ$ SP $\succ$ Pure. That means that a Remote_Types package declaration can only depend on other Remote_Types, Shared_Passive and Pure units.

The example shown on figure 2(a) highlights several DSA features. This system is based on a storage and a set of factories and workers. Each one of these entities is a partition itself. A factory **hires** a worker from a pool of workers and **assigns** a job to him. The worker performs the job and **saves** the result in a storage common to all the factories. The worker **notifies** the factory of the end of his job.

The worker produces a result corresponding to a query. When needed, a factory consumes this result. To do this, we define a protected area in the SP package Storage (sample 2(d)). An entryless protected object ensures serialized access on this area.

Types is a Remote_Types package that defines most of the remote services of the above system (sample 2(f)). First, we define a callback mechanism used by a worker to notify the end of his job to a factory. This is implemented using RAS *Notify*.

We define an abstract tagged type *Worker* which is intended to be the root type of the whole distributed worker hierarchy. *Assign* allows a factory to propose a job to a worker and a way to notify its employer the end of this job. *Any_Worker* is a **remote access to class wide** type (RACW). In other words, it is a reference to a distributed object of any derived type of Worker class.

G1_Worker is derived from type Worker and Assign is overridden (sample 2(c)). Sample 2(e) shows how to derive a second generation of workers G2_Worker from the first generation G1_Worker. As mentioned above, this RT package can be duplicated on several partitions to produce several types of workers, and also workers at several remote locations.

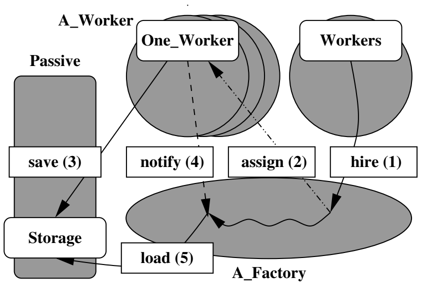
In sample 2(g), we define a unique place where workers wait for jobs. *Workers* is a Remote_Call_Interface package with services to hire and free workers. Unlike Remote_Types packages, Remote_Call_Interface packages cannot be duplicated.

In order to use even more DSA features, *Factory* is defined as a generic RCI package (sample 2(b)). Therefore, any instantiation defines a new factory (sample 2(h)). To be a RCI itself, this instantiation has also to be categorized.

In CORBA, the IDL is a descriptive language; it supports `C++` syntax for constant, type and operation declarations. From IDL descriptions, a translator can generate client header files and server implementation skeletons.

An IDL file starts by defining a *module*. This provides a name space to gather a set of interfaces. This is a way to introduce a level of hierarchy whose designation looks like (*<module>::<interface>::<operation>*). The Ada 95 binding maps this element into a (child) package. *#include* statements make any other name spaces visible.

A module can define *interfaces*. An interface defines a set of methods that a client can invoke on an object. An interface can also define exceptions and attributes. An exception is like a `C++` exception: a data component can be attached to it. An attribute is a



(a) Global picture

```
with Storage; use Storage;
generic
package Factory is
   pragma Remote_Call_Interface;

   procedure Notify (Q : Integer);
   pragma Asynchronous (Notify);
end Factory;
```

(b) Generic remote unit

```
with Types, Storage; use Types, Storage;
package One_Worker is
   pragma Remote_Types;

   type G1_Worker is
      new Worker with private;

   procedure Assign
     (W : access G1_Worker;
      Q : Integer;
      N : Notify);
private
   -- Declaration not shown
end One_Worker;
```

(c) First generation worker

```
package Storage is
   pragma Shared_Passive;

   protected Queue is
      procedure Save (Q, R : Integer);
      procedure Load
        (Q : in Integer;
         R : out Integer);
   private
      -- Declaration not shown
   end Queue;
end Storage;
```

(d) Shared memory data

```
with Types, Storage; use Types, Storage;
with One_Worker; use One_Worker;
package Another_Worker is
   pragma Remote_Types;

   type G2_Worker is
      new G1_Worker with private;

   procedure Assign
     (W : access G2_Worker;
      Q : Integer;
      N : Notify);
private
   -- Declaration not shown
end Another_Worker;
```

(e) Second generation worker

```
with Storage; use Storage;
package Types is
   pragma Remote_Types;

   type Notify is
      access procedure (Q : Integer);
   pragma Asynchronous (Notify);

   type Worker is
      abstract tagged limited private;
   procedure Assign
     (W : access Worker;
      Q : in Integer;
      N : in Notify) is abstract;

   type Any_Worker is
      access all Worker'Class;
   pragma Asynchronous (Any_Worker);

private
   -- Declaration not shown
end Types;
```

(f) Several DSA features

```
with Types; use Types;
package Workers is
   pragma Remote_Call_Interface;

   procedure Free (W : in  Any_Worker);
   procedure Hire (W : out Any_Worker);
end Workers;
```

(g) Static and unique workers pool

```
with Factory;
package One_Factory is new Factory;
pragma Remote_Call_Interface (One_Factory);
```

(h) Remote unit instantiation

Fig. 2. A complete DSA example

component field. For each *Attribute*, the implementation automatically creates the subprograms Get_*Attribute* and Set_*Attribute*. Only Get is provided for *readonly* attributes. An interface can derive from one or more interfaces (multiple inheritance).

The Ada 95 binding maps this element into a package or a child package. For the client stub, the implementation will automatically create a tagged type named Ref (which is derived from CORBA.Object.Ref or from another Ref type defined in another interface) in a package whose name matches the one of the interface. For the server skeleton, the implementation will automatically create a tagged type named Object (which is derived from an implementation defined private tagged type Object) in a package named Impl, which is a child package of a package named after the interface name (⟨*interface*⟩*.Impl*).

A method is defined by a unique name (no overloading is allowed) and its signature (the types of its formal parameters). Each parameter can be of mode *in*, *out* or *inout*, whose meanings are comparable to their Ada homonyms. Every exception that can be raised by a method must also be declared as part of the method signature (section 2.4 for the rationale).

The *oneway* attribute can be applied to a subprogram, giving it at-most-once semantics instead of the exactly-once default. This precludes a method from having output parameters, a return value, or raised exception. It is not portable to assume that the caller resumes its execution once the input parameters are sent.

Most CORBA data types map in a straightforward way onto predefined Ada types, with the exception of *any* and *sequence*. *any*, that can designate any CORBA type, is mapped onto a stream type with *read* and *write* operations. A *sequence* holds a list of items of a given type and is represented in Ada using a pair of lengthy generic packages. One may note that the CORBA *string* type is mapped onto the *Unbounded_String* Ada 95 type. The IDL does not provide an equivalent to unconstrained arrays.

The Ada 95 mapping provides special mechanisms to implement two difficult-to-map CORBA features. First, it provides a translation of multiple inheritance. As described above, an Ada 95 package defines a type derived from the first interface, and extends the list of its primitive operations to achieve inheritance from other interfaces (this solution is quite similar to mixins, as described in [14]). Another unnatural feature of CORBA for an Ada programmer comes from forward declarations. In Ada, two package specifications cannot "with" each others, but this can occur between two IDL interfaces. To solve this, the mapping proposes to create "forward" packages. This can result in a very non-intuitive situation where the client stub does not "with" its usual interface packages but "forward" packages instead.

When developping a distributed application with CORBA, two situations may appear. On the server side, the programmer is responsible for the IDL file. He has to understand the Ada 95 language mapping in order to avoid structures with a non-trivial implementation whenever possible, such as forward declaration and multiple inheritance. On both the server and the client side, the programmer has to deal with the generated code. A good understanding of the mapping is useful to get back and forth from the IDL file to the generated code in order to keep an overview of the distributed application. Understanding this mapping can be a tedious task depending of the host language.

IDL interface information can be stored on-line in a database called Interface Repository (IR). A CORBA specification describes how the interface repository is organized and how to retrieve information from it. The reader will note that this information is close to what the Ada Semantic Interface Specification (ASIS, see [4]) can provide.

The interface repository allows a client to discover the signature of a method which it did not know at compile time. It can subsequently use this knowledge together with values for the method's parameters to construct a complete request and invoke the method. The set of functions that permits the construction of a method invocation request at run time is the Dynamic Invocation Interface (DII).

The IR API allows the client to explore the repository classes to obtain a module definition tree. From this tree, the client extracts subtrees defining constants, types, exceptions, and interfaces. From an interface subtree, the client can select an operation with its list of parameters (type, name and mode) and exceptions.

A client has then three ways to make a request. As in the static case, he can send it and wait for the result; he can also may a one-way call and discard the result. With dynamic requests, a third mechanism is offered: the client can send the request without waiting for the result, and obtain it later, asynchronously.

The DII has a server-side counterpart, called Dynamic Skeleton Interface. Both mechanisms are powerful but very complex and tedious to use. In some respects, they also violate the Ada 95 philosophy, because strong typing is not preserved. Most users will keep working with static invocations.

2.4 Communication Subsystem

The communication subsystem is one of the key points of a distributed system: it offers basic services such as the capability to transmit a message from one part of the distributed program to another. Those elementary services are then used by higher level services to build a fully functional distributed system.

The limit between what belongs to the communication subsystem and what belongs to an external service may sometimes be difficult to draw. Moreover, something considered as a service in CORBA may be viewed as purely internal in DSA.

In the DSA world, everything that is not done by the compiler in regard to the distribution belongs to the partition communication subsystem (PCS). For example, figuring out on which partition a package that will be called remotely is located is part of the PCS's responsibility.

The PCS entry points are well defined in DSA, and described in the `System.RPC` package declaration. By looking at this package, one can notice that there is nothing related to abortion of remote subprogram calls, although the Annex states that if such a call is aborted, an abortion message must be sent to the remote partition to cancel remote processing. That means that the PCS is in charge of detecting that a call to one of its entry points has been aborted and must send such an abortion message, without any help from the compiler.

Another interesting characteristic of the PCS is its behavior regarding unknown exceptions. When an exception is raised as a result of the execution of a remote subprogram call, it is propagated back to the caller. However, the caller may not have any visibility over the exception declaration, but may still catch it with a *when others* clause. But if the caller does not catch it and let it be propagated upstream (maybe in another partition), and if the upstream caller has visibility over this exception, it must be able to catch it using its name. That means that the PCS must recognize that a previously unknown exception maps onto a locally known one, for example by being able to dynamically register a new exception into the runtime.

In CORBA, a much more fragmented approach of services was adopted: they are essentially defined externally. For example, the naming service (which maps object names to object references) is a distributed object with a standard IDL interface.

While this approach seems more pure, it has performance drawbacks. Being itself a distributed object, the naming service cannot be optimized for the needs of a specific ORB. A special case is also required in the ORB for it to be able to locate the naming service itself (chicken and egg problem): in order to get a reference on a distributed object (an IOR, Interface Object Reference) to start with, the programmer needs to have an IOR for the naming service. This IOR can be retrieved from the command line, from a file or by invoking the ORB Interface, depending on the CORBA version.

Regarding exception propagation, an ORB is not able to propagate an exception that has not been declared in the IDL interface. This restriction, although annoying because it restricts the usage of exceptions, is understandable given the multi-language CORBA approach: what should be done, for example, when a `C++` exception reaches a caller written in Ada? Note that an implementation may provide more information in the CORBA exception message, such as the `C++` or Ada exception name.

2.5 Application development

The DSA does not describe how a distributed application should be configured. It is up to the user (using a partitioning tool whose specification is outside the scope of the annex) to define what the partitions in his program are and on which machines they should be executed.

GLADE provides a Configuration Tool and a Partition Communication Subsystem to build a distributed application. The GNATDIST tool and its configuration language have been specially designed to let the user partition his program and specify the machines where the individual partitions will be executing [5]. The Generic Ada Reusable Library for Interpartition Communication (GARLIC) is a high level communication library [6] that implements the interface between the Partition Communication Subsystem defined in the Reference Manual and the network communication layer with object-oriented techniques.

The CORBA ORB provides a core set of basic services. All other services are provided by objects with IDL. The OMG has standardized a set of useful services like Naming, Trading, Events, Licensing, Life Cycle,... A CORBA vendor is free to provide an implementation of these services. The Naming Service allows the association (*binding*) of object references with user-friendly names. The Events service provides a way for servers and clients to interact through asynchronous events between anonymous objects.

2.6 Summary

CORBA provides an outstanding and very popular framework. The IDL syntax is close to `C++`. The object model is close to Java: CORBA defines only distributed objects. Furthermore, when using the Ada mapping, the stub and skeleton generated code is close to Java with two root classes, Ref for clients and Object for servers.

DSA provides a more general model. This includes distributed objects, but also regular remote subprograms and references to remote subprograms. Shared passive packages can be defined as an abstraction for a (distributed) shared memory, a persistency support or a database. Basically, the IDL is a subset of Ada 95 and the remote services are defined in packages categorized by three kinds of pragmas (RCI, RT, SP). The distributed boundaries are more transparent as the application is not split into IDL and host language sources. host languages.

In DSA, any Ada type can be used except access types, but this can be solved by providing the marshalling operations for such a type. The exception model is entirely preserved. Overloading is allowed in DSA (not in CORBA). The user can also define generic packages and use mixin mechanism to obtain some kind of multiple inheritance.

The DSA user can design, implement and test his application in a non-distributed environment, and then switch to a distributed situation. With this two-phase design approach, the user always works within his favorite Ada 95 environment. The use of pragma All_Calls_Remote also facilitates debugging of a distributed application in a non-distributed context.

To work on client stubs or server skeletons, the CORBA user will have to deal with generated code. In any case, understanding the host language mapping is always very useful. It can be required for some languages like `C++`. An Ada programmer should avoid using forward declaration or multiple inheritance (and in some respects, sequence).

The CORBA user has to re-adapt his code to the code generated by the translator from the IDL file anytime the latter is modified. He also has to use the predefined CORBA types instead of Ada standard types; he has to call ORB functions or a naming service to obtain remote object references.

As Ada 95 is its own IDL, the user does not deal with any generated stub or skeleton code. The configuration environment takes care of updating object, stub and skeleton files when sources have been updated. The system automatically provides some naming functions like declaring RCI services. It also takes care of aborting remote procedure calls, detecting distributed termination, checking version consistency between clients and servers, and preserving and propagating any remote exception.

The RM does not require a DSA implementation to work on heterogeneous systems but GLADE, like any reasonable implementation, provides default XDR-like marshalling operations. This feature can be inhibited for performance reasons. An ORB is required to implement a Common Data Representation (CDR) to ensure safe communications between heterogeneous systems.

CORBA is a very rich but very complex standard. Its drawbacks include the high learning curve for developing and managing CORBA applications effectively, performance limitations, as well as the lack of portability and security [13]. These drawbacks are the price to pay for language interoperability, a facility the Ada 95-oriented DSA does not provide.

Interoperability between compilers is not yet an issue with DSA because there is only one implementation available (GLADE). But it is a validation requirement to permit a user to replace his current PCS with a third-party PCS. We can note this issue was not resolved in CORBA until revision 2.2. For the same reasons, we can expect future DSA implementations to ensure PCS compatibility.

Using its IDL, the OMG has described a number of *Common Object Services* (COS) [8] that are frequently needed in distributed systems. Unfortunately, these specifications are limited to IDL descriptions, and most of the semantics are up to the vendor. The DSA misses such user-level libraries, including basic distributed software components. More generally, the lack of component libraries has always been a problem for Ada.

Implementing CORBA services as native Ada 95 distributed objects, taking advantage of the standard language features, yields a simpler, easy to understand and use specification. We have already implemented the Naming service, the Events service and a service close to the Concurrency one with DSA. Developping the CORBA services was an interesting experience. We realized that although those services are nicely specified by an IDL file, their semantics is quite vague in such a way portability is dramatically broken. This work will be described in a future paper.

Another major goal of the GLADE team is to export DSA services to the CORBA world. The idea is to translate all DSA features to equivalent IDL features using ASIS. This would allow a DSA user to connect his DSA server to an ORB. This would also allow applications written in other languages to invoke DSA features. We are also seeking to use this approach to offer a DII mechanism for DSA (subsection 3.4).

3 DSA services for CORBA users

3.1 Objective

Services implemented as RT or RCI packages can currently be invoked only from other Ada 95 code using the DSA mechanisms: remote procedure calls and distributed objects. This may be considered a drawback by software component developers when they consider using the DSA to implement distributed services, because this limits the scope of their products to Ada 95 application developers.

In order to promote the use of Ada 95 as a competitive platform for the creation of distributed services, we aim at providing a means for CORBA applications to become clients of DSA services. This requires:

- an automated tool to generate an IDL specification from a DSA package declaration; this specification shall be used by CORBA client developers to generate stubs for calling the services exported by the DSA package;
- any necessary interface code to map CORBA requests onto Ada primitive operation invocations; this code shall be replicated on each program partition that instantiates the DSA package, so that its distributed object instances can receive method invocation requests from the CORBA software bus.

3.2 From DSA specification to IDL file

ASIS [4] is an open, published, vendor-independent API for interaction between CASE tools and an Ada compilation environment. ASIS defines the operations such tools need to extract information about compiled Ada code from the compilation environment.

We are seeking to generate an IDL interface specification from a DSA package declaration using the ASIS API and an ASIS-compliant compilation environment. The ASIS interface allows the tool developer to take advantage of the parsing facility built in the compiler; it provides an easy access to the syntax tree built by the compiler from the package specification.

We will use it to obtain a list of Remote Access to Class-Wide (RACW) types declared in Remote_Types packages. These types are the Ada 95 constructs that denote distributed objects. We will then retrieve the attributes and primitive operations of the corresponding classes, and translate them into IDL interface specifications, with one IDL module corresponding to one DSA package, and one IDL interface corresponding to one RACW. To the extent possible, we will do so in accordance with the standard mapping between Ada and CORBA constructs.

We seek to define a mapping of all languages constructs permitted in a RCI or RT package to IDL constructs. We will develop an automated tool that performs the translation of a DSA package declaration to an IDL interface specification; ASIS will be used to inter-operate with a compliant compilation system. Independence between our tool and the underlying environment implementation will thus be assured.

3.3 Binding DSA entities with CORBA

In the previous section we exposed how we sought to produce an IDL interface definition from the declaration of a DSA package. In order to allow CORBA clients to make calls to DSA services, we also have to provide a mechanism to translate CORBA requests to DSA method invocations at run-time.

To this effect, we are considering a two-step approach. We will first generate a CORBA server skeleton in Ada using traditional ORB software: using an IDL definition generated by the automated tool described in 3.2 and a preexistent IDL translator, we will produce a CORBA skeleton in Ada. We will fill in this skeleton by implementing CORBA request handling as invocations of primitive operations on DSA objects. Each instance of a Remote_Types package will be accompanied with an instance of the skeleton code, and will act as a CORBA server: it will provide access to its local DSA object instances for all clients on the CORBA software bus (figure 3).

We will therefore have to map the DSA addressing space (RACW i. e. fat pointers — section 2.3) into the CORBA addressing space (CORBA object references). Each DSA object instance will thus be referable in the CORBA address space; clients that want to obtain services from the DSA objects will send requests to the associated CORBA objects, in accordance with the generated IDL definition.

We seek to provide an automatic code generation tool that will produce "glue" code to map CORBA requests to DSA object primitive invocations. The necessary adaptation code to interface with the underlying ORB shall also be provided; this includes a mapping of Ada data types to CORBA types, including a translation of DSA fat pointers into CORBA object references.

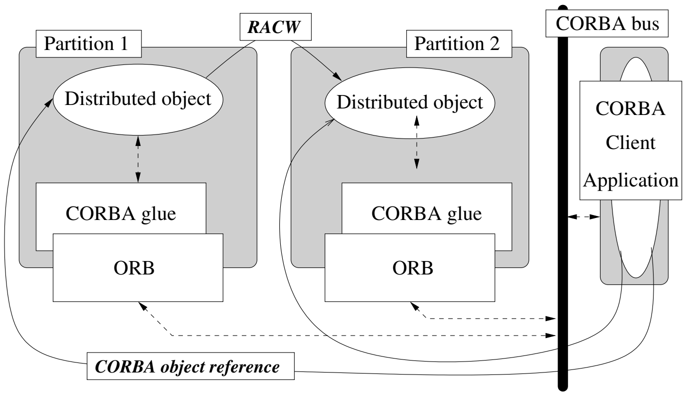


Fig. 3. Calling DSA objects from CORBA

3.4 DII using DSA

CORBA's Interface Repository service allows clients to retrieve a method's signature at run time from an interface repository, and to invoke that method even though its specification was unknown to the client at compile time using the DII mechanism (see 2.3). The DSA does not define a similar mechanism. However, a similar service can easily be provided using an RCI package that will act as a DSA interface repository, an ASIS tool that will automate the registration process of offered services, and a utility package for dynamic request construction.

The interface repository will be a straightforward RCI server that offers two kinds of services. For DSA service providers (i. e. other RCI or remote types packages), it will provide a means to register an interface, comprising a set of primitive operation names and their signatures. For clients, it will offer a means of retrieving the description of the primitive operations of a distributed object, given a reference to this object.

ASIS greatly simplifies the registration process on the server side: using ASIS, one can automatically traverse the specification of a DSA server package, and generate the corresponding calls to the interface repository's registration function. After this registration process is completed, the interface is known to the repository. In the case of the registration of the primitive operations of a distributed object (designated by a RACW), for example, any client that obtains an access value designating an object of this type can retrieve the description of its primitives even though it knew nothing of it at compile time, and does not semantically depend on the server's specification.

A function will finally construct a request message from an interface description retrieved from the repository, and actual parameters provided by the client. This message will be sent to the server using the PCS, just like a normal remote call generated by the compiler in a "static" service invocation. Apart from the generated registration functions, no IR or DII-specific code is required on the server side; it should be noted in particular that from the server point of view, a dynamically constructed request is treated exactly in the same way as a traditional, static request. The dynamic interface discovery and invocation mechanisms are confined in the DSA interface repository and the client dynamic request construction library.

This system is going to be implemented at ENST in the next few months; DSA users will thus gain the same flexibility with dynamic invocation as is currently available to CORBA programmers.

4 Conclusions

We have offered a comparison of the CORBA and Ada Distributed Systems Annex models for distributed heterogeneous applications. We have described a future tool for exporting DSA services to the CORBA world. Our general objective is to propose the interesting features of CORBA to DSA users.

Acknowledments: The authors would like to thank Brad Balfour (Objective Interface) and Frank Singhoff (ENST) for their fruitful comments. They also would like Top Graph'X for their free release of ORBAda.

References

1. A.D. Birell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
2. Andrew Birell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. Technical Report 115/94, Digital Systems Research Center, February 1994.
3. ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC 8652:1995.
4. ISO. *Information Technology – Programming Languages – Ada Semantic Interface Specification (ASIS)*. ISO, 1999. ISO/IEC 15291:1999.
5. Yvon Kermarrec, Laurent Nana, and Laurent Pautet. GNATDIST: a configuration language for distributed Ada 95 applications. In *Proceedings of Tri-Ada'96*, Philadelphia, Pennsylvania, USA, 1996.
6. Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. GARLIC: Generic Ada Reusable Library for Interpartition Communication. In *Proceedings Tri-Ada'95*, Anaheim, California, USA, 1995. ACM.
7. Sun Microsystems. *RMI – Documentation*.
8. Open Management Group. *CORBAservices: Common Object Services Specification*. Open Management Group, November 1997. OMG Technical Document formal/98-07-05.
9. Open Management Group. *The Common Object Request Broker: Architecture and Specification, revision 2.2*. Open Management Group, February 1998. OMG Technical Document formal/98-07-01.
10. Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc, 1996.
11. Laurent Pautet and Samuel Tardieu. Developping distributed Ada application in Ada 95. In *Tutorials of Tri-Ada'97*, Saint-Louis, Missouri, 1997.
12. Douglas C. Schmidt and Steve Vinoski. Comparing alternative client-side distributed programming techniques. *SIGS C++ Report*, Col 3:1–9, May 1995.
13. Douglas C. Schmidt and Steve Vinoski. Comparing alternative server programming techniques. *SIGS C++ Report*, Col 4:1–10, October 1995.
14. Alfred Strohmeier, Stéphane Barbey, and Magnus Kempe. Object-oriented programming and reuse in Ada 9X. In *Tutorials of Tri-Ada'93*, volume 2, pages 945–984, Seattle, Washington, USA, September 1993.