CORBA and CORBA Services for DSA
Extended abstract — For review only

Abstract

Comparing CORBA and the Ada95 Distributed Systems Annex shows that an advantage of CORBA is its Common Object Services, providing standard frequently-used components for distributed application development. This paper presents our implementations of similar services for the DSA. We also introduce new developments of our team that aim at providing close interaction between CORBA and Ada applications.

1 Introduction

A software developer who wants to create a distributed heterogeneous application faces a difficult choice. Several object models and protocol suites are available for this purpose, each with its own advantages and particular features; they are not currently interoperable. In this paper we more specifically focus on two particular architectures: OMG CORBA and the Ada 95 Distributed Systems Annex (DSA).

CORBA[1] is sponsored by the Open Management Group, a consortium of software vendors that seek to promote industrial standards for the development of distributed heterogeneous applications. It is based on OMG IDL, an interface description language whose syntax is close to C++. The object model is close to Java, only allowing the definition of distributed objects. The CORBA standards also define mappings of IDL into host languages such as C++, Java and Ada 95. Client stubs and server skeletons in host language are automatically generated by an IDL compiler, and they interface with a communication subsystem, the Object Request Broker (ORB), through a vendor-specific API. An ORB uses a set of standard protocols to communicate with its peers. CORBA thus permits interoperation of clients and servers that are independently coded in different languages, and using different vendors' products.

The Ada 95 Distributed Systems Annex is part of the Ada 95 ISO standard [4]. It aims at providing a framework for programming distributed systems within the Ada language, while preserving its strong typing. Its distributed application model is more general, as it can not only include distributed objects, but also remote subprograms (providing a classical remote procedure call facility) and references to remote subprograms. It also allows the definition of a shared data space, through the abstraction of Shared passive packages. In the case of DSA, the IDL is not a separate language (as in CORBA), but the host language itself. This affords developers an integrated approach for application development and test: going from a non-distributed application which is easy to test and debug to a full distributed system only requires the addition of one categorization pragma to each package that defines remote objects or subprograms. The Remote Call Interface (RCI) categorization pragma makes the subprograms of a package available for remote procedure calls, while the Remote Types (RT) pragma allows access-to-class-wide types declared in the package to designate remote objects. Such access types are then called RACWs (Remote Access to Class-Wide).

We have developed an implementation of the Distributed Systems Annex for the GNAT compiler[6]; the details of our comparison of DSA and CORBA features can be found in [8]. In this paper we first discuss the implementation of common services for the DSA. Theses services provide functionalities that are frequently required in distributed applications, and are potentially useful to all DSA developers. We then describe our implementation of an Ada IDL precompiler and CORBA binding. These are the necessary tools for creating Ada software that will interoperate with CORBA clients and servers. We finally present a new project of our team: an automated tool to make the functionality of a DSA server available to CORBA clients. This allows a server implementor to only code a DSA server, while being able to provide the same service to clients from the DSA and CORBA worlds.

2 CORBA common services for DSA

The CORBA ORB provides a core set of basic communication services. All other distributed services that an application may use are provided by objects described with IDL contracts. The OMG has standardized a set of useful services like Naming, Concurrency, Events, Trading, Licensing, Object Life Cycle, etc. A CORBA vendor is free to provide an implementation of these services. It is unfortunate that the DSA currently does not provide such a set of commonly-used services. We have consequently started to design and implement a set of DSA services that provide similar functionality for DSA application developers. In this section, we also describe our current development of a dynamic invocation facility for DSA, which is quite similar with CORBA's Dynamic Invocation Interface (DII).

2.1 The Naming service

It is impractical for users of distributed applications to deal directly with representations of object references, because these are machine-oriented identifiers that designate a particular object instance at a particular physical location, without any consideration for the user-defined semantics of the object. The Naming service allows the association (*binding*) of an object reference with user-friendly names. A name binding is always defined relative to a *naming context* wherein it is unique.

A naming context is an object itself, and so can be bound to a name in another naming context. One thus creates a *naming graph*, a directed graph with naming contexts as vertices and names as edge labels. Given a context in a naming graph, a sequence of names can thus reference an object. This is very similar to the naming hierarchies that exist in the Domain Name System and the UNIX file system.

A typical usage scenario consists in obtaining a well-known remote reference that designates the naming context corresponding to the "root" of a naming hierarchy, and then executing recursive naming operations on this hierarchy. The Trading Service provides a higher level of abstraction than the Naming Service: if the Naming Service can be compared to the White Pages, the Trading Service can be compared to the Yellow Pages, allowing a user to query objects by their properties rather than by their name.

The CORBA naming services is defined as IDL module CosNaming (see code sample 1). This module defines two data types: *name component*, and *name*, which is a sequence of name components. This module also supplies two interfaces: *Naming Context* and *Binding Iterator*. The Naming Context interface provides the necessary operations to bind a name to an object, and to resolve (look up) a name in order to obtain the associated object reference. The Binding Iterator interface is used to walk through a collection of names within a context; such a collection is returned by the *list* operation of the Naming Context interface.

Translating the CosNaming service definition to Ada using the standard mapping is not trivial; CosNaming makes use of three OMG IDL features that

```
module CosNaming {
  typedef string Istring;
  struct NameComponent {
    Istring id;
    Istring kind;
  };
  typedef sequence <NameComponent> Name;
  enum BindingType {nobject, ncontext};
  struct Binding {
    Name binding_name;
    BindingType binding_type;
  };
  typedef sequence <Binding> BindingList;

  interface BindingIterator;

  interface NamingContext {
      exception CannotProceed {
          NamingContext cxt;
          Name rest_of_name;
        };
      void bind (in Name n, in Object obj)
        raises (CannotProceed);
      void list
        (in unsigned long how_many,
         out BindingList bl,
         out BindingIterator bi);
      // Other declarations not shown
    };

  interface BindingIterator {
      boolean next_n
        (in unsigned long how_many,
         out BindingList bl);
      // Other declarations not shown
    };
};
```

Sample 1: CosNaming IDL

are not easily represented in Ada: sequences, exceptions with members, and forward interface declarations. Excerpts of the generated code are given in samples 2 and 3.

We have implemented a similar service with native Ada95 distributed objects. We were thus able to take advantage of standard language features; this yields a simple specification, which is far easier to understand and use than the CORBA one (see samples 4 and 5).

2.2 The Events service

The Events service provides a way for servers and clients to interact through asynchronous events between anonymous objects. A *supplier* produces events, while a *consumer* receives event notifications and data. An *event channel* is the mediator between

```
with Corba.Object, Corba.Sequences,
   Corba.Forward;
package CosNaming is

   type Istring is new Corba.String;
   type NameComponent is
   record
      id : Istring;
      kind : Istring;
   end record;

   -- Example of a sequences mapping
   package NameComponent_Unbounded is
      new Corba.Sequences.Unbounded
         (NameComponent);
   type Name is
      new NameComponent_Unbounded.Sequence;

   package BindingIterator_Forward is
      new Corba.Forward;
end CosNaming;
```

Sample 2: CosNaming

consumers and suppliers. *Consumer admins* and *supplier admins* are in charge of providing *proxies* to allow consumers and suppliers to get access to the event channel (dashed arrows in figure 1). Suppliers and consumers produce and receive events through their associated proxies (see plain arrows in figure 1). From the event channel point of view, a *proxy supplier* (or *proxy consumer*) is seen as a consumer (or a supplier). Therefore, a proxy supplier (or proxy consumer) is an extended interface of consumer (or supplier). The Events service defines *push* and *pull* methods to exchange events. This allows to define four models to exchange events and data.

We have developed an Events service for the DSA. During the implementation of the service, we realized that although the service is nicely specified by an IDL file, most of its semantics are quite vague; the behaviour of some methods is vendor-dependent in such a way portability is seriously compromised. Other CORBA services also suffer similar vagueness in definition. For this reason, we decided to implement only the Naming and Events services as defined by OMG, and to implement other services directly as Ada units with well-defined semantics (see 2.3).

Note that `Proxy_Push_Consumer` defined in `Event_Channel_Admin` inherits from `Push_Consumer` defined in `Event_Communication` (sample 6). The

```
with Corba.Object, Ada.Exceptions;
use CosNaming, Ada.Exceptions;
package CosNaming.NamingContext is

   type Ref is new Corba.Object.Ref with
      null record;
   function To_NamingContext (
      Self: in Corba.Object.Ref'class)
      return Ref'class;

   -- An IDL exception is mapped to an
   -- Ada exception plus a tagged record.

   CannotProceed : exception;
   type CannotProceed_Members is
      new Corba.Idl_Exception_Members with
   record
      ctx : NamingContext;
      rest_of_name : Name;
   end record;

   function Get_Members (
      X: in Exception_Occurrence)
      return CannotProceed_Members ;

   procedure bind (Self: in Ref;
      N: in Name;
      Obj: in Corba.Object.Ref);

   -- Forward reference to BindingIterator.
   procedure list
      (Self : in Ref;
       how_many: in Corba.Unsigned_Long;
       bl: out BindingList;
       bi: out BindingIterator_Forward.Ref);

   -- [some declarations are missing]
end CosNaming.NamingContext;
```

Sample 3: CosNaming.NamingContext

OMG has extended this service to provide typed data operations. An Ada 95 programmer would easily adapt our implementation by using stream operations to get this new service.

2.3 The Mutex service

CORBA defines a Concurrency service, that basically offers a complete locking system to serialize concurrent access to a resource. Extended features such as "intent to lock" are also defined by this service.

We have chosen to implement basic locking services using a more decentralized approach. Our service is based on a distributed algorithm described by Li and Hudak in [7], which avoids using a central lock manager. It has been described in [9], while a previous implementation can be found in [3].

```
package GLADE.Naming is
   pragma Remote_Types;

   type Istring is private;
   function Get_Istring
     (I : in Istring) return String;
   procedure Set_Istring
     (I : in out Istring; S : in String);

   type Name_Component is record
      Id, Kind : Istring;
   end record;
   type Name_Component_Sequence is
      array (Natural range <>)
     of Name_Component;
   type Name is private;
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Naming;
```

Sample 4: GLADE.Naming

2.4 Dynamic invocation in 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 Dynamic Invocation Interface mechanism. The DSA does not define a similar facility. However, such a 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 operations of a distributed object, given a reference to this object.

ASIS [5] is an open, published, vendor-independent API for interaction between CASE tools and an Ada compilation environment. It defines the operations needed by such tools to extract information about compiled Ada code from the 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

```
with GLADE.Objects; use GLADE.Objects;
with GLADE.Naming; use GLADE.Naming;
package GLADE.Naming.Interface is
   pragma Remote_Types;

   type Binding_Iterator is
      tagged limited private;
   type Binding_Iterator_Ref is
      access all Binding_Iterator'Class;

   type Naming_Context is
     new Objects.Object with private;
   type Naming_Context_Ref is
      access all Naming_Context'Class;

   procedure Bind
     (Ctx : access Naming_Context;
      N   : in Name;
      Obj : in GLADE.Objects.Object_Ref);
   procedure List
     (Ctx      : access Naming_Context;
      How_Many : in  Natural;
      BL       : out Binding_List;
      BI       : out Binding_Iterator_Ref);
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Naming.Interface;
```

Sample 5: GLADE.Naming.NamingContexts

built by the compiler from a compilation unit.

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 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 operations 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 Partition Communication Subsystem (PCS), just like a normal remote call generated by the compiler in a "static" service invocation. Apart from the generated registration functions, no Interface Repository 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.

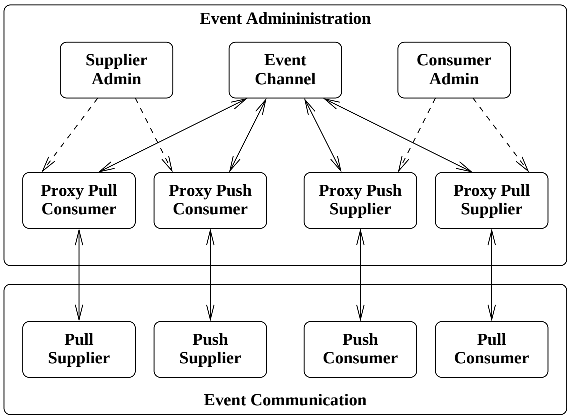

Figure 1: Structure of the Events service IDLs

This system is going to be implemented by our team in the next few months; DSA users will thus gain the same flexibility with dynamic invocation that is currently offered to CORBA programmers.

3 A CORBA ORB for Ada

The CORBA standard specifies a mapping of IDL to Ada. Using an IDL to Ada precompiler and the corresponding ORB, it should be possible to implement CORBA clients and servers in Ada. Unfortunately, we do not know of any existing free, open-source implementation of such tools. We feel that this situation makes it impractical to evaluate CORBA with Ada during a project's prototyping phase, to integrate them in a critical application where source code availability is required, or to use them in an educational context.

However, free C and C++ ORBs with C and C++ IDL precompilers were readily available. We have therefore decided to develop an Ada binding for a

```
with Ada.Streams; use Ada.Streams;
package GLADE.Event_Communication.Interface is
   pragma Remote_Types;

   type Push_Consumer is
     abstract tagged limited private;
   type Any_Push_Consumer is
      access all Push_Consumer'Class;

   procedure Disconnect
     (Consumer : access Push_Consumer)
      is abstract;
   procedure Push
     (Consumer : access Push_Consumer;
      Event    : in Stream_Element_Array)
      is abstract;
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Event_Communication.Interface;
with GLADE.Event_Communication.Interface;
use  GLADE.Event_Communication.Interface;
package GLADE.Event_Channel_Admin.Interface is
   pragma Remote_Types;

   type Proxy_Push_Consumer is
     abstract new Push_Consumer
     with private;
   type Any_Proxy_Push_Consumer is
      access all Proxy_Push_Consumer'Class;

   procedure Connect
     (Consumer : access Proxy_Push_Consumer;
      Supplier : in Any_Push_Supplier)
     is abstract;
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Event_Channel_Admin.Interface;
```

Sample 6: GLADE Event Interfaces

free ORB's internal API, and to implement our own IDL precompiler targeted at Ada 95 using this API (see [2]). We selected the C++ ORB omniORB[1] for this project. This ORB, available from AT&T Laboratories Cambridge (formerly ORL) under the GNU General Public License, provides a fairly complete implementation of the CORBA standards and of the C++ language mapping, and has proven extremely performant, particularly under Linux.

omniORB's IDL to C++ precompiler is based on the free Sun IDL front-end. We have developed a new back-end targeted at Ada 95, and integrated it in omniORB's precompiler. Our tool complies with the OMG standard Ada language mapping, and generates client stubs and implementation skeletons in Ada. We also have implemented Ada packages that encapsulate the transport facilities of omniORB. The C++ omniORB library provides two classes that correspond to different views of CORBA objects: `Object`, which is the ancestor class for all server implementations, and `omniObject`, which embodies the network resources associated with an object. Our Ada binding encapsulates `omniObject`, and reimplements a native `Object` class entirely in Ada, thus allowing us to have a clean, well-defined interface between the generated code and the underlying ORB functionality, and to limit the scope of our dependance on a specific ORB implementation.

We have thus effectively provided a free, open-source implementation of an IDL to Ada precompiler, and of matching ORB libraries. Our work is based on omniORB, and will be freely available and redistributable.

[1]For detailed information about omniORB, see `http://www.uk.research.att.com/omniORB/omniORB.html`

4 A CORBA interface for DSA services

4.1 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 dispatching operation invo-

cations; 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.

4.2 From DSA specification to IDL file

We are seeking to generate an IDL interface specification from a DSA package declaration using the ASIS API and an ASIS-compliant compilation environment. We use ASIS to walk the syntax tree of a DSA package declaration produced by the compiler. We have defined a formal mapping of all legal constructs in such a package to IDL entities. From the DSA syntax tree, we construct and IDL syntax tree. For example, for each tagged private type declared in the DSA package, an IDL interface is created, with the type's primitive operations mapped to the interface's operations. To the extent possible, this mapping is done in accordance with the standard mapping of IDL constructs to Ada [1].

We provide an automated tool that performs the translation of a DSA package declaration to an IDL interface specification; ASIS will be used to interoperate with a compliant compilation system. Independence between our tool and the underlying environment implementation will thus be assured.

4.3 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 4.2 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 an RT 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 2).

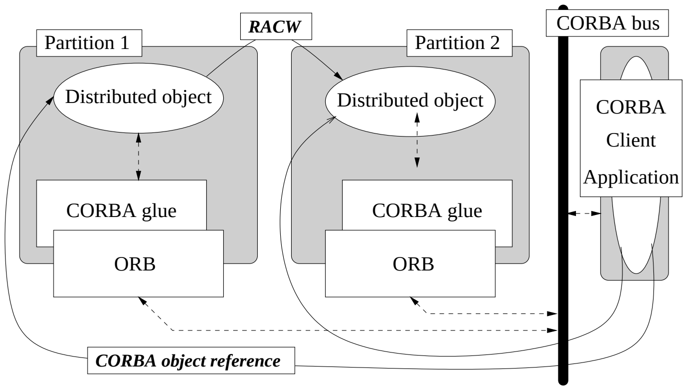

Figure 2: Calling DSA objects from CORBA

We will have to provide a CORBA address (called an IOR, Interface Object Reference) for each DSA object. This will be achieved by registering each DSA object instance on the fly with the underlying ORB the first time it is to be transmitted to a CORBA peer. 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.

NOTE FOR REVIEWERS
The section on DSA Services for CORBA clients will be expanded and completed with our experience of actually implementing the described software. The implementation is planned to be completed by the end of summer 1999.

5 Conclusion

CORBA defines a set of useful services for development of distributed heterogeneous applications. We offer a specification and implementation of similar services of general interest for users of the Distributed Systems Annex of Ada 95. We also provide an implementation of a free, open-source CORBA ORB for Ada using the omniBroker ORB. We finally offer a mapping of the DSA object model to OMG IDL, and an automated tool to make a DSA service available to CORBA clients.

References

[1] *The Common Object Request Broker: Architecture and Specification, revision 2.2.* February 1998. OMG Technical Document formal/98-07-01.

[2] Fabien Azavant, Jean-Marie Cottin, Laurent Kubler, Vincent Niebel, and Sébastien Ponce. AdaBroker, using OmniORB2 from Ada. Technical report, ENST Paris, March 1999.

[3] R. Berrendonner, L. Bousquet, T. Quinot, and S. Thellier. Mururoa: Distributed mutual exclusion using distributed objects in Ada 95. Technical report, ENST Paris, December 1997.

[4] ISO. *Information Technology – Programming Languages – Ada.* ISO, February 1995. ISO/IEC/ANSI 8652:1995.

[5] ISO. *Information Technology – Programming Languages – Ada Semantic Interface Specification (ASIS).* ISO, 1998.

[6] Yvon Kermarrec, Laurent Pautet, Gary Smith, Samuel Tardieu, Ron Thierault, and Richard Volz. Ada 95 Distribution Annex Implementation for GNAT. Technical report, Texas A&M University, College Station, Texas, USA, April 1995. (Contract with a grant from Computer Sciences Corp.).

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[8] Laurent Pautet, Thomas Quinot, and Samuel Tardieu. CORBA & DSA: Divorce or Marriage? In *Proceedings of AdaEurope'99*, Santander, Spain, June 1999.

[9] Laurent Pautet and Samuel Tardieu. Building fault tolerant distributed systems using IP multicast. In *Proceedings of SigAda'98*, Washington, DC, USA, November 1998.