A Comparison of CORBA and Ada's Distributed Systems Annex

Andrew Berns
Department of Computer Science
University of Northern Iowa
Cedar Falls, IA 50614-0507
+ 1.319.521.7663
adberns@gmail.com

ABSTRACT

With the recent advent of affordable computers, and the increased computing needs in numerous different fields, distributed computing has become quite popular. Along with this popularity has come several different approaches for creating distributed systems. Ada is fortunate to have multiple approaches available, each distinctively different from each other. One approach is to use a separate middle layer to allow different parts of the Ada distributed program to communicate. The Common Object Request Broker Architecture, or CORBA, uses this approach. Another approach is to use an extension of the language made for the creation of distributed systems, meaning no extra layers need to be added. This language extension approach is used by Ada's Distributed Systems Annex. While both approaches have their merits, CORBA has received much more attention than Ada's Distributed Systems Annex. This paper is the result of an undergraduate research project to examine how easy each approach was, and to see if the extra attention given to CORBA is deserved.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Classifications – *concurrent, distributed, and parallel languages.*

General Terms

Performance, Design, Languages.

Keywords

CORBA, Distributed Systems Annex, distributed computing.

1. INTRODUCTORY INFORMATION

1.1 Corba

CORBA first came on the distributed scene when it was created by the Object Management Group (OMG) in 1991, and was attractive to many because of its combination of object-oriented design features (such as polymorphism, inheritance, and encapsulation), and its transparent communication layer to make distribution easier for the programmer and allow language interoperability [2]. In the short time CORBA has been in existence, it has attracted a large market, with numerous language mappings and different vendor implementations available. The wide variety of products available is a testament to CORBA's increasing popularity.

CORBA attempts to create programmer-transparent distributed systems by going through several layers in its internal architecture. At the heart of CORBA is an Object Request Broker (ORB). The ORB handles all interaction between distributed objects, as well as provides other extra services. Each CORBA implementation available is another implementation of the ORB and subsystems. While some ORBs contain extra features, all CORBA-compliant ORBs adhere to the OMG specification – although there is still room for differences between ORBs [8].

The ORB knows which objects accept what messages through the use of CORBA's Interface Definition Language (IDL) [9]. IDL is a simple language that developers use to define what their object interfaces will be – items such as what methods they respond to, what variables they allow access to, and what types they define (such as records and enumeration types). IDL is relatively simple to learn because it contains only syntax for defining objects, not implementing them. Also, IDL is kept relatively small, as it attempts to find a shared subset of language features from many different languages.

While IDL is used to define the interface to the objects, most of the work in a CORBA system occurs elsewhere. An IDL compiler creates client stubs and server skeletons in the implementation language (Java, Ada, C++, etc.), and the programmer fills out the implementation file for the object. This language-specific implementation is called a servant [1]. The implementation code uses the skeleton code created with the IDL compiler, as well as a Portable Object Adapter (POA – an object that allows multiple implementation languages in a single CORBA system), to communicate with the ORB [2]. Clients who wish to use the object call the compiler-created stub code, which communicates with the ORB and sends the message to the appropriate object implementation, or servant.

How the ORB transmits the messages it is to delegate to the different objects is mostly left up to the implementation. The CORBA standard defines a General Inter-ORB Protocol (GIOP), which is the interface that the ORB is to use to communicate.

However, implementations of this GIOP can use any network protocol or communication mechanism they wish [6]. The typical instance of the GIOP is the Internet Inter-ORB Protocol (IIOP), which uses TCP/IP to transmit messages. Several other protocols exist, however, including a Multicast Inter-ORB Protocol (MIOP), which uses multicast IP, and a Unix Inter-ORB Protocol (UIOP), which uses Unix sockets. This avoidance of defining exactly how objects communicate leads to a lot of flexibility, allowing the appropriate communication mechanism to be chosen for the specific application.

All these objects work together to create a distributed system. Below is a sample diagram that shows two client objects communicating with a single servant:

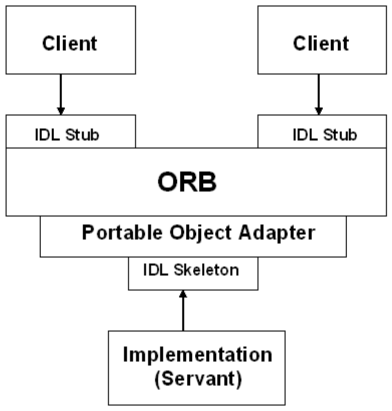

Figure 1. Sample CORBA Program Architecture

Because the ORB handles communication, it can offer several advanced services as well. Perhaps one of the more beneficial ones is the ability to dynamically reference objects and interfaces – that is, an object need not know at compile-time exactly which object it will be talking to, but can find this at run-time [1]. The complicated design of the CORBA architecture allows features such as these to exist.

To create a CORBA program, the developer must bring together many of these steps through several conceptually simple tasks. First, the interfaces for the distributed objects must be created using IDL. Once the necessary interfaces have been created, the IDL compiler takes the IDL files and creates stub code for the clients, and skeleton code for the servers. After this is complete, programs to create and initialize the ORB and server objects must be created. Client code can then be created, which accesses the ORB created inside the server program. The client code will use the POA inside the ORB to get references to the distributed objects, and then communicate with them. Obviously, using this system allows a large amount of language interoperability, but also requires a few extra files to "plug in" the new implementation to the ORB.

1.2 Ada's DSA

Ada's Distributed Systems Annex (also known as Annex E) takes a different approach from CORBA. Ada's DSA creates a distributed program using the existing Ada language, without the use of an intermediary language such as IDL. This allows the freedom to use most of Ada's features, but limits language interoperability. The DSA broadly defines how a distributed system should be created, but the tools and mechanisms to do so are defined by the implementation, allowing plenty of freedom in exactly what takes place inside the distributed system.

For the Distributed Systems Annex, distribution is handled through the interaction of several partitions using one of several different mechanisms. A partition is the entity that the developer uses to denote a "distributed piece" of the program. A partition can contain one or more library units, which are selected after program creation. All library units inside a partition exist within the same address space, and communicate without the use of a distributed interface [4]. Distributed interaction occurs between partitions, which can exist either on many different nodes or the same node. DSA allows the developer to specify which library units it wishes to be distributed and which not to be by placing them inside partitions.

There are two types of partitions – active and passive [3]. An active partition is allowed to have one or more threads of control, such as a task. Active partitions cannot share data directly with one another, but rather must use remote calls. Passive partitions are not allowed to have a thread of control, but are allowed to provide shared data that active partitions can access. Passive partitions allow active partitions a way to communicate without continual message passing. Regardless of whether a partition is passive or active, however, it is not a first class type inside Ada – meaning it cannot be created dynamically, but rather must be bound at the time of the creation of the distributed application [4].

Partitions communicate through the use the Partition Communication Subsystem, or PCS [4]. Ada defines the specification of the PCS, but does not provide an implementation. Therefore, each tool that implements the DSA is allowed to provide their own mechanism for communicating between partitions. Like CORBA, this means that any method could be used to pass data, not just one, such as TCP/IP or UDP. This means the possibility exists for a great variety in communication mechanisms and performance between vendors who implement the Distributed Systems Annex.

Inside each partition is a number of library units. These library units are the packages and subprograms that will be used inside the distributed system. Partitions are allowed to contain units that do not communicate with other distributed partitions. These are called normal units, and must be duplicated on each partition in which they are needed [5]. Units that wish to participate in the distributed interactions are labeled with a variety of pragmas, specified by the Ada language specification. These pragmas denote how the library unit will participate in the distributed system, as well as enforce all the necessary restrictions on such a unit. Through the use of pragmas, the developer can select which units they wish to use for their distributed system.

The Distributed Systems Annex allows a distributed unit to be assigned one of three types: Remote Call Interface, Shared Passive, and Remote Types. The remote call interface unit (assigned by pragma Remote_Call_Interface [5]) specifies that the package is available to receive subprogram calls from other partitions, representing the typical remote procedure call paradigm for distributed systems. Like other communication aspects of the distributed system, calls to the remote procedure are done just as if it were non-distributed. A remote call interface package can exist on only one partition, and all other partitions receive a stub to forward the calls on to the main package. Remote calls can be done both statically (as in a typical, non-distributed system), or they can be done dynamically through the use of remote access to subprogram types (RAS) and remote access to class-wide types (RACW) [7]. An RCI package has several limitations, such as variables may not be declared inside the specification, and there can be no non-access type declarations without providing special mechanisms for communicating the type.

The shared passive unit (assigned by pragma Shared_Passive [5]) is a unit that is used for shared storage throughout the system. A shared passive unit can exist on either an active or a passive partition, although it is not allowed to have any threads of control. It is useful for a shared file or storage space approach, as all partitions using the shared passive partition will have access to the same data. However, as shared passive units are not allowed to have threads of control, they have heavier restrictions, such as not allowing tasking, and only allowing entryless protected objects.

Finally, remote type units (assigned by pragma Remote_Types [4]) define those types that shall be used across the entire system, especially the definition of classes for distributed objects. As these types must be shared across the distributed system, only access types are allowed to be used for remote types. These access types store both the local address of the data, and also the address of the node, and because of this, they are called “fat pointers”. Remote type units must be pre-elaborable packages, and thus are not allowed to contain any variable declarations inside the specification.

Ada also uses several pragmas not related to DSA system configuration. Pragma Pure declares a package without any variables or access types – that is, a package with a constant state. Pragma Asynchronous is used to denote that a procedure should be executed asynchronously (control returned to caller before the procedure is done executing). Pragma All_Calls_Remote is useful for debugging RCI packages, as it requires all calls to go through the PCS, even if the calls are local [4]. While not specifically related to unit assignment, these pragmas are useful for compile-time design decisions inside an Ada distributed program.

In summary, DSA uses categorized library units inside specified partitions that communicate through the PCS to create distributed programs. This method is quite easy to understand and requires little thought towards distribution on the part of the application developer. Below is a diagram of three sample partitions communicating through Remote_Call_Interface and Shared_Passive units:

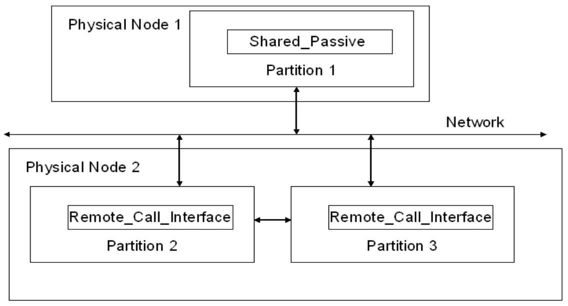

Figure 2. Sample DSA Program Architecture

Creating a distributed application using Ada's DSA is quite simple. First, the developer can create the application as if it were not distributed. Programs that can be distributed through DSA can also be compiled and executed outside of DSA, offering several advantages, such as quicker testing and debugging, as well as offering an alternative for when multiple nodes are not available. While almost any program can be distributed using DSA, it is a good idea for the developer to keep in mind the eventual distribution during program design by creating packages with loose coupling (of course, this is good programming practice, anyway). After the program has been created, it can be partitioned by using a tool that comes with the implementation of the DSA. After partitioning, the program simply needs to be executed on the node or nodes that have been configured. The creation of a distributed Ada program is quite similar to the development of a non-distributed Ada program.

2. SOFTWARE TOOLS

As mentioned before, CORBA is very popular, and there are many different ORB implementations to choose from, each with their own individual advantages and drawbacks. The implementation that was chosen for this comparison was PolyORB, an implementation from AdaCore. PolyORB's General Inter-ORB Protocol (GIOP) has several instances: Internet Inter-ORB Protocol (IIOP), which uses TCP/IP; Multicast Inter-ORB Protocol, which uses multicasting; and UDP Inter-ORB Protocol (DIOP), which uses UDP to communicate [6]. For this paper, only the IIOP GIOP was used.

Ada’s Distributed Systems Annex, like CORBA, allows implementation developers plenty of freedom in selecting exactly how a system should be created. Surprisingly, however, there are few DSA configuration and partitioning tools available. For this comparison, GLADE, also from AdaCore, was chosen, as it is the oldest and most widely known DSA tool available. To partition and configure the system, the developer supplies GLADE with a fairly simple configuration file, and GLADE does the rest. GLADE also includes several other features, such as automatic starting of the partitions and data compression filters for the network [5], which were not tested.

3. DISTRIBUTED TEST APPLICATIONS

3.1 Echo Program

To test the two different methods of creating a distributed system, two programs were created. First, to help understand the basic concepts, and to demonstrate usability, a simple echo program was written. This program consists of a server that receives an Echo command with a string argument. The server than prints the message to standard output. The specification from the DSA package used as a Remote_Call_Interface and the IDL for the CORBA object are shown below (the EchoHello program is adapted from [6], p. 23):

```
package EchoHello is
   pragma Remote_Call_Interface;

   procedure Echo( Message : in String );
end EchoHello;
```

Figure 3. EchoHello Specification

```
interface EchoHello {
   void Echo( in string Message );
};
```

Figure 4. EchoHello IDL

3.2 Prime Number Finder

The second program designed was used to time the different distributed programs. It uses the classic computing problem of finding prime numbers. To check, it simply begins dividing a number by all the odds that are less than or equal to its square root. If it finds none that evenly divide it, the number is prime. There are much more efficient algorithms to calculate primes; however, this program is meant not to find primes, but to test the advantages of a distributed system.

The program works by having a client constantly poll a server object to see if it has been initialized (busy waiting was used because I had problems using entry calls inside CORBA, which I suspect is a problem with my configuration, not a limitation of CORBA). Once the server has been initialized, clients request an array of numbers, which are broken up using a protected object so that no two clients receive the same array. After they have found the prime numbers in their array, they pass them back to the server, which stores them inside a local variable. Thus, this test program uses a sufficient amount of data passing to test the communication systems, but also has enough computing per node to make it appropriate for distributed systems. The Remote_Call_Interface specification and the IDL for the CORBA object are shown below:

```
package PrimeFinderServer is
   pragma Remote_Call_Interface;

   type Integer_Array is array (0..199) of Integer;

   procedure Initialize( Limit : in Natural );

   function Get_Array return Integer_Array;

   function Finished return Boolean;

   procedure Put_Primes( Primes : in Integer_Array;
                         Count : in Integer );
   function Ready return Boolean;
end PrimeFinderServer;
```

Figure 5. PrimeFinderServer Specification

```
interface PrimeFinderServer {
   typedef long Integer_Array[200];

   void Initialize( in long Limit );
   Integer_Array Get_Array( );
   boolean Finished( );
   void Put_Prime( in Integer_Array primes,
                   in long Count );
   boolean Ready( );
};
```

Figure 6. PrimeFinderServer IDL

4. FINDINGS

4.1 Qualitative Findings

The first thing I noticed when I started my project was the learning curve for each tool. Having had no hands-on experience with distributed systems before beginning this project, I was able to compare how easy it was to understand each distributed system method. I found Ada's Distributed Systems Annex to be much easier to learn than CORBA, for several reasons. First, learning CORBA requires learning a new language, while learning DSA does not. While IDL is simple and enough like C++ or Java to learn relatively easy with previous exposure to these languages, it still requires things to learn that are not an issue when creating a DSA program, such as naming conventions, type conversions, and scoping [9]. Also, understanding how DSA works is much simpler than understanding how CORBA works. The only conceptual difference between a distributed and a non-distributed program with DSA is the communication protocols, which are of little concern to the average developer. However, to understand a CORBA program, many layers of complexity must be known, such as registering objects with the ORBs, and getting references to objects. This added complexity and the need for a new language makes CORBA more difficult to learn than DSA.

Secondly, creating a program in CORBA was much more difficult than doing so in DSA. A great example of this is with the PrimeFinderServer program. For the DSA version, hardly any extra work beyond creating a non-distributed version was required – simply add a pragma to a package and create a simple configuration file. For CORBA, however, several extra coding steps were required, such as creating a package for initializing the ORB and getting references to the correct objects. Each of these steps requires several extra lines of code that can be confusing to a beginner. The figures below show the major differences between the distributed and non-distributed programs for each method - the configuration file of the PrimeFinderServer DSA program, and the initialization file for the ORB (the initialization file was adapted from [6], p. 25).

```
configuration distprime is
   Starter : Partition := (PrimeFinderStarter,
                           PrimeFinderServer);
   procedure PrimeFinderStarter is in Starter;

   Client_1 : Partition := (PrimeFinderClient);
   for Client_1'Host use "illinoiscentral";
   for Client_1'Directory use "/usr/distributed/ada";

   Client_2 : Partition := (PrimeFinderClient);
   for Client_2'Host use "pennsylvania";
   for Client_2'Directory use "/usr/distributed/ada";

   Client_3 : Partition := (PrimeFinderClient);
   for Client_3'Host use "reading";
   for Client_3'Directory use "/usr/distributed/ada";

   Client_4 : Partition := (PrimeFinderClient);
   for Client_4'Host use "delawarehudson";
   for Client_4'Directory use "/usr/distributed/ada";

   Client_5 : Partition := (PrimeFinderClient);
   for Client_5'Host use "sooline";
   for Client_5'Directory use "/usr/distributed/ada";

   Client_6 : Partition := (PrimeFinderClient);
   for Client_6'Host use "greatnorthern";
   for Client_6'Directory use "/usr/distributed/ada";

   procedure PrimeFinderClient;
   for Client_1'Main use PrimeFinderClient;
   for Client_2'Main use PrimeFinderClient;
   for Client_3'Main use PrimeFinderClient;
   for Client_4'Main use PrimeFinderClient;
   for Client_5'Main use PrimeFinderClient;
   for Client_6'Main use PrimeFinderClient;
end distprime;
```

Figure 7. GLADE configuration file for PrimeFinderServer

```
with Ada.Text_IO;
with CORBA.Impl;
with CORBA.Object;
with CORBA.ORB;
with PolyORB.CORBA_P.CORBALOC;

with PortableServer.POA.Helper;
with PortableServer.POAManager;

with PrimeFinderServer.Impl;

with PolyORB.Setup.No_Tasking_Server;
pragma Elaborate_All( PolyORB.Setup.No_Tasking_Server);

-- modified from sample file included inside
-- PolyORB User's Guide v 1.2r, Jerome Hugues,
-- page 25
procedure PrimeFinderStarter is

begin
   declare
      Argv : CORBA.ORB.Arg_List :=
     CORBA.ORB.Command_Line_Arguments;
   begin
      CORBA.ORB.Init(
     CORBA.ORB.To_CORBA_String("ORB"), Argv );

      declare
         Root_POA : PortableServer.POA.Ref;
         Ref : CORBA.Object.Ref;
         Obj : constant CORBA.Impl.Object_Ptr :=
       new PrimeFinderServer.Impl.Object;
      begin
         Root_POA := PortableServer.POA.Helper.To_Ref
                (CORBA.ORB.Resolve_Initial_References
            (CORBA.ORB.To_CORBA_String("RootPOA")));
         PortableServer.POAManager.Activate
            (PortableServer.POA.Get_The_POAManager
       ( Root_POA ) );

         Ref := PortableServer.POA.Servant_To_Reference
             (Root_POA, PortableServer.Servant(Obj));

         Ada.Text_IO.Put_Line("'" &
      CORBA.To_Standard_String(
       PolyORB.CORBA_P.CORBALOC.Object_To_Corbaloc(Ref))
      & "'" );

         CORBA.ORB.Run;
      end;
   end;
end PrimeFinderStarter;
```

Figure 8. CORBA PrimeFinderStarter

A note should be made, however, that while the CORBA ORB initialization is quite large, once it is running, nodes can be dynamically added, while the DSA program requires the nodes to be enumerated at configuration time, meaning if more nodes are to be added, more lines must be added to the configuration file.

Perhaps the biggest advantage CORBA has over Ada's Distributed Systems Annex is language interoperability. Because CORBA offers ORBs that are compatible with many different languages, a developer using CORBA has present and future flexibility to modify the system as needed, selecting the language or languages that they find best suitable for the system. The DSA, however, only allows distributed pieces to be written in Ada, meaning present and future flexibility is limited by the features of the Ada programming language. This becomes a large disadvantage when certain features are required, such as graphics, that are easier to implement in a language other than Ada. Using the Distributed Systems Annex gives up the flexibility that language interoperability offers.

Another area where CORBA is ahead of the Distributed Systems Annex is availability of documentation. As CORBA is quite popular and available in many different language implementations, there is a large amount of material concerning CORBA, ranging from ORB architecture to program design and creation. While literature on CORBA with Ada is not as available as other languages, such as Java and C++, much of the CORBA information can be used for any language, meaning only a little Ada-specific documentation is needed. Ada's DSA, however, has comparatively little published about it. The largest amount of information I obtained concerning DSA program creation was taken from the GLADE user's manual [5]. However, because using Ada's DSA is only slightly more complex than creating a normal program in Ada, the limited availability of adequate documentation only slowed down program development a small amount.

Finally, I found development easier with Ada's Distributed Systems Annex because I could compile and run the program without thinking about it being distributed. A DSA program can run non-distributed just as easily as distributed, and so I did not need to worry about valid network references, working communication subsystems, or even having the distribution tools. I could develop the program on a completely separate computer, with or without pragmas (the pragmas were accepted on multiple compilers), and when it was completed, I could then worry about distributing it, knowing that the program worked. CORBA, on the other hand, required me to have the ORB available on all computers I wished to develop on. Simple test runs still required the overhead of the full program, and debugging the program required considering both the CORBA statements inside the program, as well as the Ada code. Programs written using DSA were easier to compile and debug than distributed programs built using CORBA.

4.2 Quantitative Findings

Below are the results from 2 different sets of testing, both on clients for the prime number finder. For the first test, multiple client programs were executed on the same physical node to see how the different implementations handled non-distributed execution, which can become important if a large project needs to be developed where the distributed environment is not available.

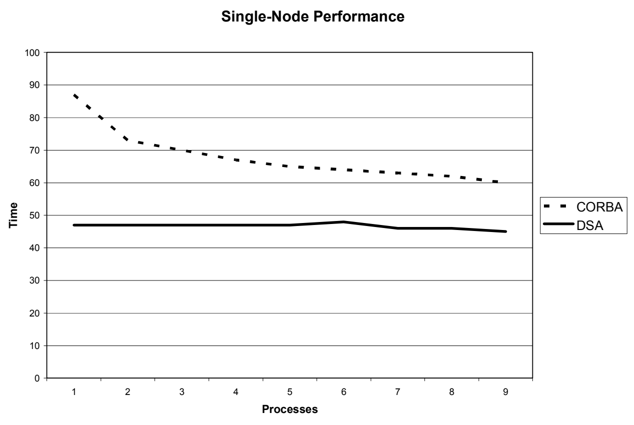

Figure 9. Single-Node Performance Comparison

The interesting thing concerning the single-node performance is that Ada's DSA kept a consistent time throughout, no matter how many client processes it had running. Conversely, CORBA was able to decrease by a small amount its computational time by increasing the number of nodes. A client process inside DSA would use as much processor as it could (for a single node, over 90%). However, a CORBA node would only use about 50% of the processor for the tests. This result was surprising, as it seems to suggest that for programs that need to be tested on a single machine, Ada's Distributed Systems Annex offers a more consistent performance than CORBA.

The second set of tests checked how the system performed as more and more physically-separate clients were added. This provides a picture of how the communication subsystems react to increased usage, as well as the benefits of distribution in general.

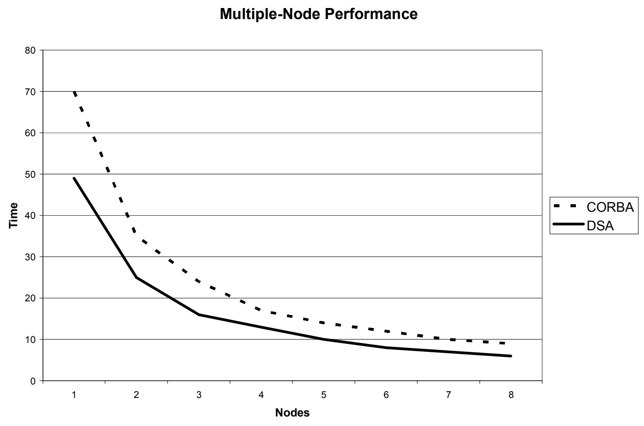

Figure 10. Multiple-Node Performance Comparison

Here, there was much less difference in between implementations. CORBA and DSA performed almost identically in regards to relative improvement as more nodes were added. That is, CORBA and DSA respond the same way to an increase in processing nodes – there is no performance differences arising from their communication subsystems in this particular problem. It would appear from these tests that Ada's DSA and CORBA are both similarly equipped to handle distributed program communication through TCP/IP amongst different nodes.

5. CONCLUSION

While CORBA is a powerful tool that has many uses, DSA seems to have enough advantages in certain areas over CORBA that I believe it should have at least some areas where it is more suitable for distributed systems than CORBA is. For instance, a program could be created and verified by a team of developers, and then later pragmas added to distribute it, meaning a very small amount of extra effort is needed to distribute the program (even for existing systems created without distribution in mind), and a much smaller testing effort would be required after distribution. A CORBA application, however, would seem to require more effort earlier on in the life cycle to create a distributed system, and more training and education for developers. Also, a distributed program can be created with only a minor amount of education beyond Ada, meaning for schools or companies already using Ada, distributed systems could be taught much easier than with CORBA. Also, given Ada's excellence in real-time systems, DSA could be a great way to create distributed real-time systems without extensive redesign or paradigm shifts. There are many potential uses of Ada's Distributed Systems Annex that would be interesting to explore in more detail.

Overall, I found that Ada's DSA seems to be worthy of much more attention than it receives. Because it is easy to learn, use, understand, and debug, while at the same time performing equal to or better than similar CORBA implementations, Ada's DSA seems to have the potential for a large market amongst any developers interested in creating distributed systems with Ada. The small amount of tools and support available were the only downsides I could find from my preliminary examination of the Distributed Systems Annex. Hopefully such a powerful and promising tool will continue to be developed and gain a wider acceptance, as the Distributed Systems Annex surely deserves it.

6. REFERENCES

[1] Baker, S. *CORBA distributed objects using Orbix*. Addison-Wesley, Harlow, England, 1997.

[2] Brose, G., Vogel, A., & Duddy, Keith. *Java programming with CORBA*. John Wiley & Sons, Inc, New York, 2001.

[3] Burns, A., and Wellings, A. *Real-Time systems and programming languages*. Addison-Wesley, Harlow, England, 2001.

[4] Burns, A., and Wellings, A. *Concurrency in Ada*. Cambridge University Press, Cambridge, UK, 1998.

[5] *GLADE user's guide*. Retrieved electronically on March 8, 2005 at http://libre.act-europe.fr/glade/.

[6] Hugues, J. *PolyORB user's guide*. Retrieved electronically on March 2, 2005, http://polyorb.objectweb.org/download.html 2004.

[7] Kermarrec, Y. CORBA vs. Ada 95 DSA: A programmer's view. In *Proceedings of the 1999 annual ACM SIGAda International conference on Ada.* (Redondo Beach, California, October 17-21, 1999). ACM Press, New York, NY, 1999, 39-46

[8] Mowbray, T. J., and Ruh, W. A. *Inside CORBA: Distributed object standards and applications*. Addison Wesley Longman, Inc, Reading, MA, 1997.

[9] Object Management Group. *Ada language mapping specification*. Retrieved electronically on April 5, 2005 from http://www.omg.org/technology/documents/formal/ada_lang uage_mapping.htm, 2001.