Inside the distributed systems annex

Laurent Pautet and Samuel Tardieu

École Nationale Supérieure des Télécommunications
46, rue Barrault
F-75634 Paris Cedex 13, France

Abstract. The current Ada revision tries as much as possible to provide the programmer with an easy way to build distributed systems; in many cases, the programmer can easily modify its monolithic application and transform it in a very short amount of time into a distributed one. However, there is a huge amount of work involved in the compiler and in external tools and libraries to build and run a distributed application without loosing Ada semantics and strong type checking. This paper describes how such a task has been achieved in the current implementation of GNAT, the GNU Ada compiler, and in GLADE, GNAT's companion package for building and running distributed programs.

1 Annex E: a short overview

This section describes the key features of the distributed systems annex. The reader interested in the other features should read the full text in [12].

1.1 Distributed systems

A *distributed system* is a computer program that executes on several processing units at the same time. This partitioning across several processors and hosts may be necessary because of the following points (non exhaustive):

CPU power requirements: if the program is computation intensive, then a single processor may not be sufficient to achieve its goals in a timely fashion

CPU specialization: some processors may be unable to perform efficiently complex mathematic computations while some others will be unable to directly access certain devices

Fault tolerance: it may be required that some parts of the program be duplicated, in order to provide an uninterrupted service despite computer crashes and network temporary unavailability

Repartition of the application on various sites: a travel agency for example will use large databases replicated on several secondary servers that are not too far from terminals located across the country to ensure a decent response time

In Ada 95, a distributed program is a regular computer program in the sense that it has one entry point (the main procedure) but may be separated in a semi-transparent way into *partitions*. The partitions will then communicate with each other by exchanging data, using remote subprogram calls and distributed objects, a much more powerful mechanism than the traditional message-passing mechanism used in client/server applications (message-passing is equivalent to *goto*s in non-distributed applications while remote subprogram calls are similar to regular subprogram calls).

1.2 Streams manipulation

Basic distributed systems are easily built using the facilities offered by the underlying operating system, e.g. BSD sockets (see [11]). However, these tools operate only on arrays of bytes without any consideration of their content. To write portable distributed programs that can run on heterogeneous architectures, the programmer must carefully choose the byte ordering (big-endian *vs.* little-endian), the word size and the floating point format.

In order to work around this difficulty, Ada provides the user with the possibility to define four attributes (*Input*, *Read*, *Output* and *Write*) that control how a value will be transformed into and from a freely chosen "stream format". All the standard types already have implementation-defined subprograms for these attributes, but there is no guaranty that the default values are suitable for heterogeneous processing. These attributes are inherited by types derived from the standard types unless they are overloaded by the user. Also, these attributes do have pre-defined values for compound types such as records or arrays.

To build a distributed system that will run on different architecture, one can adopt an encoding method such as XDR (see [6]). However, this does not solve the problem of access values, that are meaningful only on one partition since this is likely that another partition will have a completely different object space filled in with different objects. The user still needs to define its own Read and Write attributes to read and write the designated value rather than the pointer itself.

Also, some data structures do not need to be transmitted as-is. For example, transmitting a complete graph to another partition only requires that the vertices be transmitted; since the full structure can be rebuilt at the other end, sending the edges would add zero information (using the definition from [10]). Some other data structures require extra care too, for example doubly linked lists, not to run into an infinite loop while flattening the elements.

1.3 The remote subprogram call paradigm

As we wrote in section 1.2, it is possible to write a distributed system based on low-level communication primitives. However, even if we consider the endianness problem and friends as solved, the programmer still faces the slight incompatibilities between various interfaces to the low-level routines. For example, reading or writing several file descriptors at a time may require that the user uses "select" on some systems and "poll" on some others.

The distributed systems annex allows the programmer to forget about these details. Strong data typing is preserved, including types of class-wide objects that can still control dispatching operations after being transmitted across partitions; the exception model is not modified in any kind and localizing and contacting other partitions as needed is taken care of at run time. The *remote subprogram call* paradigm which is the most intuitive one for the programmer is being used, while basic communication systems provide the user with a message-passing only mechanism. A remote subprogram call is no more no less than a regular subprogram, except that it may require that the network be accessed. It is a two way process from the caller's point of view: it makes a call and waits for the answer or the exception to come back.

As an extension, some procedures may be tagged as *asynchronous* (using the "Asynchronous" pragma). An asynchronous procedure is a procedure which has only "in" or "access" parameters. These kind of procedures do not respect the Ada semantics because they have two properties:

1. They may return before the completion of the procedure
2. Any exception raised during the procedure execution is lost (this property is a side effect of the first one)

Using asynchronous procedures, it is very easy to build an application that is deterministic when executed in non-distributed mode and non-deterministic in distributed-mode, by using for example nested asynchronous procedures whose execution path can follow different network links with different speeds. However, these procedures are very useful in some types of applications. For example, sending an informational message to a user's terminal need not raise an exception if the user's partition is unreachable, and the operation of sending such a benign message should not block the whole application for a long time, even if the underlying network is slow.

1.4 Categorization pragmas

Annex E of the Ada Reference Manual (see [12]) introduces three new categorization pragmas whose goal is to identify packages that must have a "remote-aware" property. These pragmas are:

pragma Remote_Types: a package containing this pragma will be used to define types that can be called remotely. These types must have some characteristics ensuring that they can be transmitted between partitions running on various nodes without loosing their semantic meaning. Basic types such as Integer, Character, Float do have an intrinsic meaning that cannot be loosed while exchanged between partitions. On the other hand, pointers may be meaningless outside of the partition where they have been built; if the user wants to transmit a pointer type across partitions, he has to provide *Read* and *Write* attributes that will encode the information in a meaningful way. For example, if a linked list is to be transmitted between partitions, the user can define attributes that will flatten the list and build an array before the transmission, while a linked list structure will be rebuilt from the array on the receiver side

pragma Remote_Call_Interface: this pragma identifies a package that will offer remote subprograms to the rest of the application. Any subprogram defined in the visible part of such a package will result in a network access if necessary. The user needs not be aware of whether such an access will occur or not; its effect must be transparent and semantically equivalent to a regular procedure call[1]. Also, *remote access to subprogram* types can be defined in these packages to define dynamically bound remote subprograms

pragma Shared_Passive: packages that are said to be Shared_Passive contain data that can be shared amongst partitions running on the same processing node. This data can be seen as regular shared memory

1.5 Distributed objects

The concept of *distributed objects* is a very popular paradigm; the well-known CORBA technology, amongst others, uses them as the basis for building distributed systems. The power of dynamic dispatching combined with the Distributed Systems Annex makes distributed objects an appropriate solution for highly dynamic and evolutive distributed systems.

With distributed objects, existing code can be extended without modification or recompilation since calls to primitive operations ("methods" in some non-to-be-cited programming languages) will automatically dispatch on the new object's code if needed.

In Ada, a special class of pointers are dedicated to distributed objects; they designate any object in a hierarchy rooted by a (possibly abstract) tagged limited private type. These pointers are called *remote access to class-wide types* (or *RACW*).

The declaration of the designated type and of its primitive operations, which must occur in the visible part of a Pure or Remote_Types package declaration, looks like this:

```
type Object is tagged limited private;
procedure Primitive (O : access Object);
```

while the declaration of the remote access to class-wide type reads:

```
type RACW is access all Object'Class;
```

and must be located (paragraph E.4 (18) of the Reference Manual) into the visible part of a Remote_Types or Remote_Call_Interface package declaration.

[1] The semantic equivalence cannot always be strictly guaranteed given that the subprogram body can have access to low-level information concerning the partition it is running on; also, some characteristics such as the time used to perform the subprogram call will not be identical.

Thereafter, if a statement “Primitive (R);” appears with R being of type RACW, the call is not only dispatching on the tag of the object designated by R, but also on the partition where the access value has been created.

1.6 A standardized interface

As the vast majority of operating systems do not include transparent distributed processing, a *Partition Communication Subsystem* interface has been specified in the reference manual by means of a language-defined package named *System.RPC*. This package defines useful types such as *Partition_ID* which represents a partition identifier in a distributed program, as well as key subprograms such as *Do_RPC* which is the entry point for a remote subprogram call.

Despite this standardization effort, a lot of things have been left up to the implementation, for example the way a distributed application is divided into partitions or where and how the various partitions are launched. In the rest of this paper, we will try to show how the annex specification and much more were implemented; for example, some of the concepts defined in CORBA (see [8]) or in RMI (see [5]) have been mapped onto corresponding features, to take the best side of every world.

2 GLADE: an implementation

GLADE (see [1], [4] and [7]) is an implementation of the distributed systems annex for GNAT. It is jointly developed by Ada Core Technologies and the ENST in Paris, France. It comes with its own implementation of System.RPC, which fulfills the requirements of the reference manual. That is, another compiler implementing correctly the Distributed Systems Annex should be usable with GLADE.

2.1 Structure of GLADE

GLADE is divided in two major parts:

GARLIC is the partition communication subsystems. It is made of:
- **System.RPC,** a compliant implementation of the package described in the Reference Manual
- **System.RPC.*,** child packages of System.RPC, that provide additional services such as message passing, as allowed by the Reference Manual
- **System.Garlic** and child packages, the heart of the partition communication subsystem, which takes care of network-related system calls, concurrent requests, partition localization and launching, error handling and recovery, etc., used by System.RPC
- **System.Stream_Attributes,** a XDR implementation of standard stream attributes (see section 1.2) to adapt streams to an heterogeneous environments[2]

[2] GLADE 2.02 will offer a choice between XDR, CDR (see [8]) and possibly other encoding methods such as DER/ASN.1 (see [2]).

GNATDIST is the partitioning tool, responsible for:

– checking the consistency of a distributed system before building it
– calling GNAT with the appropriate parameters to build the needed stubs
– configuring the filters that will be used between different partitions
– linking the partitions with GARLIC and building the initialization sequence
– building the main program that will launch the whole distributed application on the right hosts

2.2 Compilation model

Since some subprograms can be called remotely in a distributed system, the traditional compilation model which consists of the generation of an object file from a source file cannot be applied anymore. At least two object files must be generated, one for the calling side and the other for the receiving side. This model is very close to what is done in CORBA and RMI, except that in Ada the user does not need to write extra code or use another IDL[3] (Ada is used as the IDL) to describe the signatures of the remote subprograms or objects.

Up to version 3.10, GNAT was generating two files (called stubs files) containing legal Ada code, one for the calling stubs, one for the receiving stubs. These stubs files had to be compiled to produce object files. However, this model, despite its simplicity and ease of use, was not suitable for all the constructs; some complex structures could not be expressed in legal Ada without using lots of unreadable code. As of version 3.11, GNAT generates object files directly thus decreasing the processing time since only one pass is necessary whereas two passes were required in the past. The user interested in reading the content of the stubs can still use the appropriate compiler flags to see the expanded re-generated Ada-like tree.

3 Implementing remote calls

3.1 General case

Let's consider a function Add with the following profile:

function Add (X, Y : Integer) **return** Integer;

We will examine what needs to be done to perform this function call, in both non-distributed and distributed cases.

Non-distributed case The various steps performed by the executable when this function is called in the non-distributed case are listed below:

1. Put X on the stack[4]

[3] Interface Definition Language
[4] "stack" is a general term here, since on some architectures the first parameters are passed in registers that emulate a stack.

2. Put Y on the stack
3. Call function Add
4. Read the return value from a dedicated hardware register

If an exception has been raised in the function's body, then the execution will never reach point 4, but will be redirected onto the exception handler instead.

Distributed case: caller point of view The calling stubs will contain some code looking like the following:

1. Look for the partition containing function Add and put the RPC_Receiver corresponding to the package containing Add in the stream
2. Put the index of the function Add in a stream
3. Put X in the stream
4. Put Y in the stream
5. Send the stream to the receiving partition
6. Get the result stream back
7. Get the exception occurrence from the result stream
8. If it is not Null_Occurrence, then re-raise the exception occurrence
9. Read the result (sum of X and Y) from the result stream, it is the return value of the function

Distributed case: receiver point of view The receiving stubs will include:

1. Get the RPC_Receiver from the stream, and jump to the corresponding dispatching procedure (which will execute stages 2 to 5 below)
2. Get the function index from the stream
3. This function takes two integers, so read two integers X and Y from the stream
4. Call the Add function (which is present on the current partition) with parameters X and Y
5. If an exception occurred, put the exception occurrence into the result stream and send it back; otherwise, put Null_Occurrence then the return value in the result stream and send it back

These steps are not performed by the environment task of the receiving side. Instead, a new task called "anonymous task" is created to serve the request. That way, a potentially unlimited number of requests can be handled at the same time.

This scheme shows that the Ada model has been completely preserved. If an exception is raised during the execution of Add, then it is re-raised in the calling partition, thus interrupting the right flow of control. Also, the user does not have to take care of putting the parameters with the correct format in the stream, because the compiler itself will generate the right sequence using its knowledge of argument and return types.

While this example looks trivial, it may become much more complicated when unconstrained or tagged objects are considered. Also, “out” and “in out” parameters must be handled as if they were return values, possibly with an additional size or discriminant constraint. For these values, it is necessary to transmit the bounds of the objects while performing the call, so that the callee may check that it does not try to write past the bounds of an array for example. Also, the fact that an object that has discriminants is constrained or not must be given to the callee before it can process the request.

3.2 Asynchronous remote procedure calls

Asynchronous procedures are implemented in a straightforward way: after having put the parameters in the stream and sent the request, the caller does not wait for the result stream. The receiver will throw away any exception raised during the execution of the procedure, and will never send any stream back to the caller.

3.3 Handling remote abortion

The Ada Reference Manual states that if a construct containing a remote subprogram call is aborted, whether the processing is also aborted on the receiver end or not is implementation-defined. If we consider large applications with heavy computations, it seems desirable to abort a computation as soon as we know that its result will not be used because the caller has been aborted. For this reason, we chose to implement remote job abortion to release the CPU when possible.

To do this, GLADE associates a *request id* with every non-asynchronous remote call. This identifier is bound to the request and transmitted to the receiver side along with the request itself. Also, a controlled object is declared in GLADE within the scope of the call. If the caller is aborted, the *Finalization* procedure of the controlled object will be executed anyway, thus suspending the abortion for the time of the finalization. Then this finalization subprogram sends an abortion request to the receiver partition with the right request id. Upon reception of this abortion command, the receiver side aborts the task that was bound to this request, thus achieving the propagation of the abortion undergone by the caller to the receiver.

4 Implementing distributed objects

Because we consider distributed objects as being a very important feature for building distributed systems, we have tried to make them as efficient as possible in GNAT. One strong requirement was to avoid any overhead for objects located on the partition on which the call is made.

In order to achieve this, GNAT does not use a wrapper for all objects, but instead create a new derived object type to designate remote objects. Also, GNAT creates a wrapper for all primitive operations of the original type, hence catching all attempts to use the object[5].

```
type Remote is new Object with record
   Partition : System.RPC.Partition_Id;
   Receiver  : System.RPC.RPC_Receiver;
   Addr      : System.Address;
end record
procedure Prim (Object : access Remote);
```

The three fields respectively correspond to the partition on which the object has been created, the address of the subprogram which will handle the incoming remote calls to the object's primitive operations, and the address of the object on its own partition. Note that what is presented here is only a functional equivalence. In fact an access discriminant on such a structure is added to the object.

Read and Write attributes for the remote access to class-wide type are also defined by the compiler. The Write attribute transmits the Partition, Receiver and Addr fields of the structure (if the object is local, these fields are built dynamically at run time, otherwise they are copied from the existing ones). The Read attribute works the other way around: it reads the three fields, and check the local partition ID against the object's one. If they match, then the Addr field is used and a pointer on the real local object is returned. If they don't, then a new object containing these fields is built and returned.

5 Efficiency concerns

5.1 Anonymous tasks

In section 3.1, we wrote that anonymous tasks were created to handle incoming calls in order to serve several requests at the same time. However, this mechanism was too costly to be used as-is, and has been reworked in order to avoid dynamic creation of tasks as much as possible.

In the new model, the concept of *task pool* has been introduced. Some tasks (the exact number can be set in the configuration file) are created by a low-priority background task and put themselves in a global task pool. Whenever a request arrives, a task is extracted from the pool and assigned to this request. When the request terminates, the task inserts itself back into the pool and waits for another job.

If the number of tasks is insufficient to handle all the incoming requests, then more tasks will be dynamically created and destroyed. For this reason, it is important to have an idea of how many incoming requests may occur at the same time; a number too high will require memory space that could be used for something else, while too low a number may cause frequent dynamic task allocation.

[5] Since the object is of a tagged limited private type, the only possibility to access the object's fields is to call a primitive operation on this object.

5.2 Storage handling

Up to GLADE 1.03, all the structures used for communicating were created on the stack, as local variables. This became a problem when some users needed to transmit very large arrays which caused repetitive (and fatal) stack overflows.

The safe and low computation cost method adopted in GLADE 2.01 is to dynamically allocate the arrays of bytes needed to communicate using a special storage pool. This storage pool reserves a fixed number of data structures of a fixed size at elaboration time. Whenever an array of byte is needed, the allocator corresponding to this storage pool gets called automatically and first tries to return a pre-allocated block if the required size is not greater than the pre-allocated size, thus avoiding "real" dynamic allocation.

When such a pointer gets freed, the deallocator corresponding to the storage pool is called and if a pre-allocated block had been used, it returns to the list of free block without calling the system deallocation routine. With these constructs, dynamic memory allocations occur only at elaboration time, when no more pre-allocated blocks are available or when a request for a huge block is made, thus reducing dramatically the costs of heap allocation while guaranteeing the safety of the application by not using the stack for potentially huge allocations anymore.

5.3 Name server, location server and communication

GLADE uses two different services to locate partitions on the network:

1. A name server that maps the package name on a partition id and checks for the coherence of the versions of the various packages used in the distributed application (using a checksum-based mechanism)
2. A location server that maps the partition id onto a host

These services are separated because they will be independent in the future; in our fault-tolerant model, partitions will be replicated and the location server will get new locations for the new replicas, while dead partitions will be removed from its database. However, the distributed application structure will not change and the data on the name server will not be modified.

To avoid frequent useless exchanges between those two servers and other partitions, the results returned by these servers are cached onto each partition. If a location becomes unavailable, then the location server will either send an invalidation request to any partition that knows anything about the dead partition, or wait until the partitions ask for the new location of the dead partition.

Once a partition knows where another partition is located, it connects directly to it using a common protocol such as TCP. This direct connection model implies that there is no need to transmit partition information along with each request, unlike IIOP (see [8]), because the link is dedicated to a pair of partitions. Also, no forwarding needs to be done since the different partitions can talk together directly. Also, the user does not need to know what a name server or a location server is since everything is done in a transparent way by the compiler and the run time libraries.

6 Configuring the system

As explained in section 1.6, the Reference Manual does not describe how a distributed application should be configured; this feature is implementation-dependent. For this purpose, we have designed a tool called GNATDIST and its associated configuration language to ease the build of a distributed application (see [3] for an exhaustive description of GNATDIST functionalities).

The configuration language looks like regular Ada declarations and is intuitive for the Ada programmer. It is parsed and analyzed by GNATDIST which then builds various executables (one executable for each partition) after producing Ada packages enabling the features that have been selected by the user in the configuration file. Since GNATDIST reuses some sources from GNAT, it can access the information usually available only to the binder. In particular, it can compute the closure of the main unit and then explore the list of Remote_Call_Interface packages to check that they have been assigned to exactly one partition, as required by Ada semantics, and that child Remote_Call_Interface packages belong to the same partition as their parent.

GNATDIST provides the user with a way of using specific services, such as filters (see [9]) or alternate termination modes. For example, it is possible to specify that an encryption filter will be used between every partition, that a compression filter will be used to communicate to one particular partition located at the end of a slow link, and that an authentication filter will be used between two important partitions of the distributed application.

Also, GNATDIST tries to minimize the number of recompilations during the development cycle by using a private subdirectory to store the stubs object files and handles the coherency of this cache. Partial compilations are also possible by specifying the name of the partitions to be compiled on the command-line.

7 Work in progress and future work

We are continuing to implement new features in GNAT and GLADE and to improve the efficiency of the system. Amongst other topics of research, we can cite (by decreasing order of importance):

- Fault tolerance partition communication subsystem; this system will support crash faults as well as byzantine failures (failures that can result in a partition sending a wrong result, opposed to no result at all for crash faults) using replication and voting mechanisms
- Enhanced distributed debugging facilities
- Very light run-time for client-only partitions, without any tasking involved
- Statistical and optimization tool to help in placing partitions while respecting the constraints given by the user.
- Graphical partitioning tool
- Distributed compilation in GNATDIST.

8 Contact and software availability

- The authors can be reached using electronic mail in ENST at addresses `Laurent.Pautet@inf.enst.fr` and `Samuel.Tardieu@inf.enst.fr`
- GNAT and GLADE are both free software and can be obtained via anonymous FTP respectively at `ftp://cs.nyu.edu/pub/gnat/` and `ftp://cs.nyu.edu/pub/gnat/glade/`
- Related tools and components for GLADE and network programming in general are available for free at `http://www-inf.enst.fr/ANC/`
- Ada Core Technologies is selling support contracts for both GNAT and GLADE. You can get more information on this support at `http://www.gnat.com/`

References

1. Anthony Gargaro, Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. PARIS: Partitionned Ada for Remotely Invoked Services. In *Proceedings of Ada-Europe'95*, Frankfurt, Germany, March 1995.
2. JTC 1/SC 33. *Specification of Abstract Syntax Notation One (ASN.1)*. 1990. ISO 8824:1990.
3. Yvon Kermarrec, Laurent Nana, and Laurent Pautet. Gnatdist: a configuration language for distributed ada 95 applications. In *Proceedings of Tri-Ada'96*, Philadelphia, Pennsylvania, 1996.
4. Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. GARLIC: Generic Ada Reusable Library for Interpartition Communication. In *Proceedings of the Tri Ada conference*, Anaheim, California, 1995. ACM.
5. Sun Microsystems. *Remote Method Invocation – Documentation*.
6. Sun Microsystems. *xdr – library routines for external data representation*.
7. Daniel Neri, Laurent Pautet, and Samuel Tardieu. Debugging distributed applications with replay capabilities. In *Proceedings of Tri-Ada'97*, Saint-Louis, Missouri, 1997.
8. OMG TC Document 97-09-01. The Common Object Request Broker: Architecture and Specification Revision 2.1. September 1997.
9. Laurent Pautet and Thomas Wolf. Transparent filtering of streams in GLADE. In *Proceedings of Tri-Ada'97*, Saint-Louis, Missouri, 1997.
10. Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1963.
11. Richard W Stevens. UNIX *Network Programming*. Prentice Hall, 1990.
12. Tucker Taft. *Ada 95 Reference Manual: Language and Standard Libraries*. February 1995. ISO/IEC/ANSI 8652:1995.