PARIS - Partitioned Ada for Remotely Invoked Services

Anthony Gargaro
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-3890
USA
abg@sei.cmu.edu

Yvon Kermarrec
Télécom Bretagne
Département Informatique
Technopôle de l'Iroise
29 285 Brest Cedex
France
yvon@enstb.enst-bretagne.fr

Laurent Pautet
Télécom Paris
Département Informatique
46, Rue Barrault
75 013 Paris
France
pautet@inf.enst.fr

Samuel Tardieu
Télécom Paris
Département Informatique
46, Rue Barrault
75 013 Paris
France
tardieu@inf.enst.fr

March 21, 1995

Abstract

This paper presents our experience implementing the Ada 95 Annex E to support distributed systems using the GNAT compiler[6]. The work has been performed by a multinational team from France and the USA. The paper describes the implementation of the Partition Communication System and the required support from the GNAT compiler. In addition, extensions beyond Annex E are described that facilitate programming the next generation of distributed applications in Ada 95.

1 Introduction

A distributed system architecture comprises a network of computers and the software components that execute on the computers. Such architectures are commonly used to improve the performance, reliability, and reusability of complex applications. Typically, when there is no shared address space available to remotely-located components, components must communicate using some form of message-passing abstraction or the remote procedure call paradigm. In both instances, the corresponding support to the application is not included within the disciplined semantics of a standard language that supports type-safe object-oriented programming.

Recently academic and industrial initiatives have attempted to compensate for this lack of support by providing a variety of different solutions. These solutions range from standardizing various message-passing interfaces to language mappings for client/server interfaces. Unfortunately, these solutions often fail to balance the desirability of raising the level of distributed programming abstractions with the predictability of application performance. For example, the use of a message-passing interface may provide predictable performance but its level of abstraction is not conducive to developing reusable components. Conversely, a client/server interface, that promotes a more abstract and transparent view of remote communication, is unlikely to meet the predictable performance required by many applications.

The Ada Distributed Systems Annex, Annex E, provides a solution to programming distributed systems that balances abstraction with predictable performance. An Ada application may be partitioned for execution across a network of computers such that typed objects may be referenced through remotely-called subprograms. The remotely-called subprograms may be either statically or dynamically bound, thereby allowing applications to use either the classical remote procedure call paradigm or the increasingly popular distributed object paradigm. In this paper, the implementation of both paradigms is explained.

1.1 Objectives

Ada incorporates facilities for programming distributed systems as a consistent and systematic extension to those provided for programming non-distributed systems. Thus, the benefits of a type-safe object-oriented programming language that supports both data access synchronization and concurrency are made available for programming distributed systems. Furthermore, the language-defined interface between the compiler and the Partition Communication Subsystem (PCS) makes it possible to use different PCS implementations so as to achieve specific application performance requirements.

A principal objective of our implementation is to provide a timely demonstration of these benefits using the GNAT compiler. Since the GNAT compiler is readily available for numerous architectures our implementation is designed to be portable. This portability is achieved by secondary objectives to conform, where possible, to the POSIX standards and to structure the PCS as reusable components. In this way, we hope to promote further contributions and refinements to our implementation so that Ada will become a dominant distributed programming language for both academic and industrial applications.

1.2 Contributors

International teams are involved in this project and they have backgrounds in Ada, compilers and distributed systems: New York University, Texas A&M University, Computer Sciences Corporation and France Telecom Universities in Brest and in Paris.

1.3 Paper outline

In the first section, we present the distribution model of Ada 95 and its relations with other efforts in both academia and industry. The second section presents the extensions that have been integrated in the GNAT front-end in order to deal with the Distributed System Annex. Next, we present the interface that Ada 95 defines between the application and the communication facilities and its interests. In the remaining section, we present the global architecture of the system we have implemented. In our conclusion, we present the status of our developments and the research in progress.

2 Design Overview

2.1 Related Work

The increasing use of workstations connected to network servers has popularized distributed systems outside of the application domains that have traditionally employed distributed system architectures to meet high performance and reliability requirements. As a consequence, various approaches for programming distributed systems have emerged to support the message-passing, remote procedure call, and distributed object paradigms.

The message-passing paradigm continues to dominate in applications where improved performance is required. Recent work in developing standard message-passing interfaces such as Message Passing Interface (MPI)[12], and POSIX 1003.12/1003.21 are indicative of the interest to increase application portability without compromising performance. For example, MPI unifies various schemes and defines both the syntax and semantics of a message-passing library. In those applications where performance is not critical, approaches such as the Parallel Virtual Machine (PVM) [1] have become attractive alternatives.

The remote procedure call (RPC) paradigm[3] is a proven approach that has been embodied in the OSF Distributed Communication Environment architecture[13]. DCE provides a collection of services similar to RPCs that are to be integrated directly in the OS. This approach is a significant step towards facilitating programming distributed systems, but still concentrates on services that are separate from the programming language.

In contrast to these two approaches, 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. The OMG Common Object Broker Request Architecture (CORBA)[5] is an industrial-sponsored effort to standardize the distributed object paradigm via the CORBA Interface Definition Language (IDL). Mappings between the IDL and different programming languages allow for the paradigm to be expressed directly in the programming language. Currently, several implementations of CORBA implementations exist that map the IDL to C++.

2.2 Definitions

The distribution of an Ada program is achieved through the Ada partition model. Library units may be included in partitions and each partition may be elaborated and executed independently. Partitions may be active or passive. An active partition comprises one or more tasks, whereas a passive partition must be preelaborated and may not perform any action that requires runtime execution. Typically, a passive partition is configured to represent a shared address space accessible to one or more active partitions. The annex requires the construction and configuration of partitions for a distributed systems architecture to be specified by the implementation; our approach is discussed in section 5 of this paper.

The annex defines package categorization pragmas that allow packages to be explicitly assigned to partitions. The categorization pragmas place restrictions on the declarations in a package; in this way inter-partition references, namely invoking remote services, can be checked to ensure that they are permissible.

There are three categorization pragmas to facilitate inter-partition references:

- Shared_Passive which support the shared objects paradigm,
- Remote_Control_Interface which support the statically bound remote subprogram (procedure) call paradigm.

- Remote_Types which support the dynamically bound remote subprogram (procedure) call paradigm; i.e., distributed objects.

In addition, the annex specifies the interface between the compiler and the Partition Communication Subsystem (PCS) in package System.RPC. A part from this interface and the pragma, the implementor has a large degree of freedom. This presents an immediate benefit as the annex does not prevent the extension of the partition model; e.g., dynamically replacing failed partitions.

The Ada partition model presents several advantages which include:

- Consistency of the distributed application is preserved since all the partitions are subject to the same rules as a non-distributed application
- Strong typing is enforced on all inter-partition references
- Clear compiler and PCS independence is maintained
- Difference between the paradigms for distributed and non-distributed programming is minimized
- Applications may be partitioned for different distributed system architectures without recompilation

2.3 Design Principles

The integration of the annex in the GNAT system has been done with the following criteria:

- Modularity. We have not mixed low level communication features with the high level communication interface so that our system can be adapted for different distributed system architectures. Our approach exploits the development of reusable software components.
- Portability. GNAT targets a wide range of architectures and operating systems with the help of the gcc back-ends. Therefore, we must deal with various network interfaces and operating systems. In this context we have designed a common interface called GARLIC (Generic Ada Reusable Library for Interpartition Communication).
- Efficiency. For obvious performance purposes, we want to make direct calls to the run time low level libraries and to implement efficient algorithms. Moreover, we use protected types and requeue to reduce the cost of race-free data synchronization.

- Availability. We want to make limited modifications in the GNAT system so that it is readily available. Therefore, we have chosen an approach where code transformation (carried out by the expander) is the major issue. This approach makes it possible for other compilers to be readily adapted to make our implementation available.

- Evolution. The implementation is intended to be used in production-quality applications and to fulfill the needs of distributed system architects, designers, and programmers. What we have implemented is a compliant open implementation that may be extended to meet future requirements.

2.4 Inter-Partition Communication

The RPC paradigm implies the notion of servers and clients. The server offers its services to its environment (it can execute some code for someone else) and the clients call these services. This raises the issue of server localization: i.e., how one client determines the address of a required server. We have discarded the static solutions which are the easiest to implement but which do not fit the programmer's need. We have a dynamic approach which involves a name server that we call the partition server. When a server has terminated its elaboration, it registers itself to this partition server: i.e., it transmits its name and information to reach its services. When a client wants to reach a server for the first time, it queries the partition server and retrieves the localization information.

The above scheme is straightforward and facilitates reconfiguring distributed applications. We depict the various elements of an RPC in Figure 1. In the context of Ada, the Partition Server is also responsible for checking that the library unit consistency of the distributed application is not compromised. It is a requirement of the annex that such inconsistencies in a distributed application are detected prior to execution.

3 Stub Generation

The distribution directives are provided through categorization pragmas. Using these pragmas, the compiler determines which subprogram calls and which memory accesses may involve remote accesses. Thus, the compiler generates specific code to handle these situations and this is performed at expansion time.

The stub generation is performed by the GNAT compiler and is based on our previous work as described in [11] and [9]. The code expansion involves the generation of stubs as depicted in figure 2. A similar treatment would apply

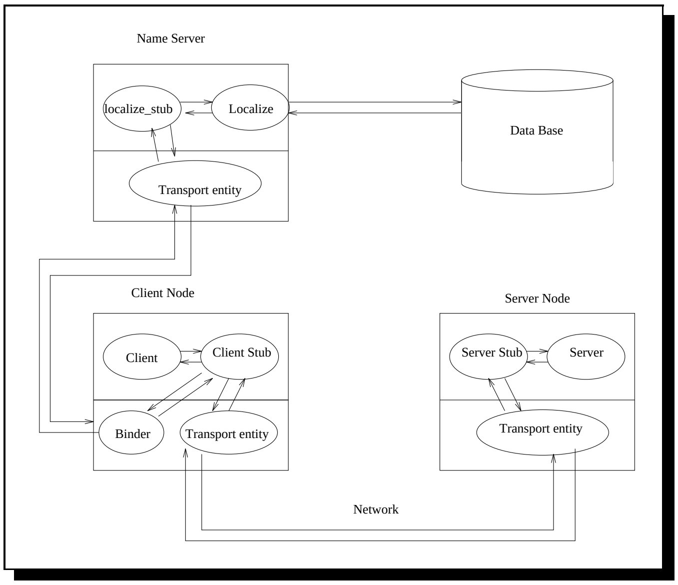

Figure 1: Remote communication architecture

in the case of programs written in IDL (interface definition language) for Sun RPC and CORBA. The code expansion performs the following steps:

- On the client side. The expanded code locates the required server, sends a request with the parameters of the RPC and waits for the result.
- On the server side. The expanded code registers the new server and makes it available to its environment, accepts requests, decodes them and executes the corresponding subprogram and then returns the result.

The complete scenario can be depicted as in figure 3.

3.1 Statically Bound Calls

This is the classical RPC operation, and presents no particular difficulties. The compiler is able to resolve the nature of the call statically and to determine the implied RCI package and the requested service.

3.2 Dynamically Bound Calls

Dynamically bound calls are integrated with Ada capabilities to dereference subprograms and to dispatch on access-to-class-wide operands. Thus their compilation requires some additional code expansion activity, beyond that required for statically bound RCI stubs.

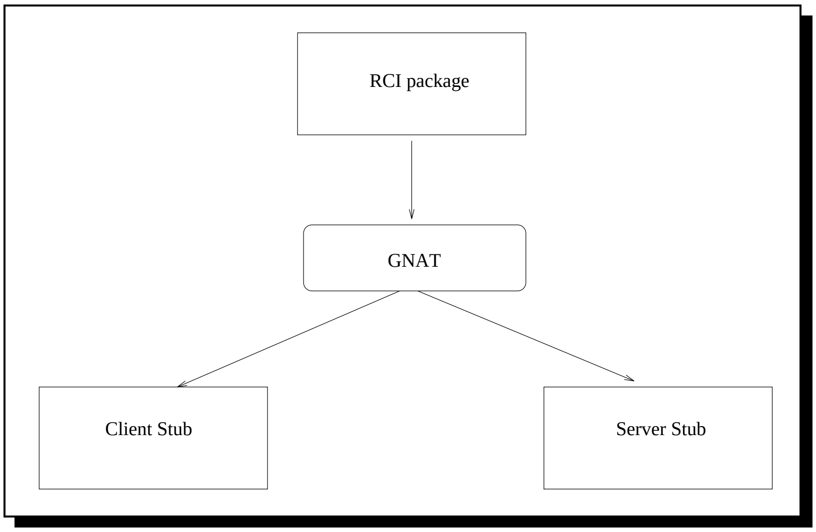

Figure 2: Code expansion in the front end of GNAT

3.2.1 Remote Access to Subprograms

For statically bound calls, the location of the service can be determined at compile time; this is not the case for remote access to subprograms. The application during execution may obtain an access value of a remote subprogram, and this access value can then be dereferenced. The previous scheme (the static one) cannot be used as neither the server partition nor the location of the service are known. Therefore, access values to remote subprograms must incorporate all the necessary information to reach their corresponding subprogram. This is accomplished by introducing the notion of a fat pointer.

The annex specifies rules that restrict the use of remote access to subprograms so as to avoid illicit usages. These rules enforce semantics that may be used by the annex implementor. We indicate the principles of the code expansion:

1. The value of an access to a remote subprogram is represented by a fat pointer, i.e., a value that includes partition and server information.

2. All operations on the original access type are replaced with operations that expect a fat pointer value. There are very few cases to consider because accesses to remote subprograms can appear only in three contexts: in calls, in assignment statements and as the expected type of the 'Access attribute. Assignments are straightforward, while the evaluation of the 'Access attribute must construct the fat pointer.

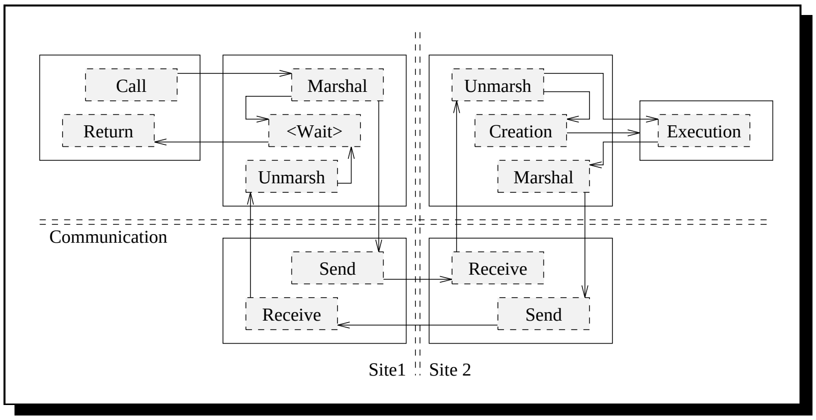

Figure 3: Involved interactions during an RPC

3. Any dereference of such a remote access value (in the context of a call) is expanded to become a remote call to the appropriate partition.

3.2.2 Remote Access to Class-wide Types

Remote access-to-class-wide types (henceforth RAC) appear at first sight to be a complex mechanism because they involve both remote operations and object-oriented features with dispatching. In that respect we introduce the notion of remote dispatching.

RAC types are introduced in the following steps. A remote types package declares an abstract type and abstract subprograms on the type, together with a RAC type designating the abstract type. The abstract type or "5 keywords" type (i.e., is abstract tagged limited private) is the root type of a class that will be used by both clients and servers. Then, an RCI server declares a set of remotely callable subprograms that allow values of the RAC type to travel among partitions. A server extends the "5 keywords" type, declares objects of that type and passes references to these objects to other partitions via the RCI server. Such references can then be used as controlling operands to call the abstract subprograms. This involves remote dispatching since the code to be executed is located on the server partition that has extended the designated type. (A complete but terse example can be found in the Ada Reference Manual.) First, a remote call is performed and then once the server partition is reached a normal dispatching call is performed.

To simplify the implementation, it is convenient to handle occurrences of RAC types as regular access types as much as possible, in order to minimize the changes to the compiled code. The technique is to define the fat pointer itself as an extension of the original abstract type, and to use references to the fat pointer to denote occurrences of the RAC in the client partition. In this way, the server code sees a local access type which references a local object, whereas the client code sees a local access type to a fat pointer that references a remote object. Some of the details follow:

1. A fat pointer is defined in the client partition. The fat pointer stores information on a remote object, that is to say its server partition and its location on that partition, in addition to other components that we do not discuss here.

2. Given that the fat pointer is an extension of the original abstract type, the subprograms of the original abstract must be overridden in the client partition.

3. Given that fat pointers are transferred from one partition to another by means of remote subprograms declared in the RCI server, 'Read and 'Write attributes must be provided for them. The 'Write operation constructs the fat pointer for the access value to the object in the server partition and the 'Read attribute recovers the fat pointer for use in the client partition.

4 System.RPC

As mentioned earlier, System.RPC is the interface to the PCS and is defined as part of the annex. The interface provides only those facilities needed to support minimal implementations; therefore, more advanced implementations are expected to provide additional facilities as child packages of System.RPC.

4.1 Read and Write

The attributes 'Read and 'Write convert values of objects into a stream of elements suitable for transmission across a communications network. Calls to these attributes are located in the client and server stubs by the expander phase of the compiler. Before sending a request to perform a remote call, the subprogram identity and parameter values must be converted by 'Write (marshaled) into a stream. This sending stream is then sent as one or more messages that are recovered into a receiving stream at the receiving partition. The receiving stream is converted by 'Read (unmarshaled) into values that are used to reconstruct a call to the subprogram with the same parameter values.

Before the stream is transformed to or from a message format, any necessary encoding or decoding of stream elements to support heterogeneous communication may be performed. For example, the stream may be encoded into a neutral network representation such as ASN.1. In this way partitions executing on different computer systems may be supported since the Read and Write subprograms implemented by System.RPC in each system may recognize the neutral network representation. Currently, we do not support such a representation in our implementation.

4.2 Do_RPC and Do_APC

These two interfaces are both related to calling a remote service. The client stub calls them and supplies three parameters: the partition identity of the server, and two streams. The first stream includes information to reach the requested service and the parameters of the call; the second stream provides a location for returning the parameters (or result) of the call, or an exception occurrence.

Both interfaces require support from communication services that are provided through other lower level interfaces within the PCS. Since Do_RPC is a blocking operation, it is necessary to ensure that only the calling task is suspended while the remote subprogram is executed. This is achieved by queuing the calling task to wait for the subprogram to complete and for the corresponding notification to be returned. Immediately the notification is returned, the calling task is dequeued and the Do_RPC completes its execution.

4.3 Establish_Rpc_Receiver

The annex leaves the role of this procedure open to interpretation by implementations. It must be called once the partition has completed its elaboration so that a connection between the PCS and the stubs of the RCI servers is established. Unfortunately, this means that it is impossible to call a server before its partition has elaborated completely. This is a potential inconvenience since mutual dependencies between RCI servers that are placed on distinct partitions are subject to deadlock condition.

Our approach is more flexible and interprets the annex to allow mutual dependencies. As soon as a RCI server completes its elaboration, it registers itself to the partition server; i.e., it may be called from other partitions. (To comply with the annex, we may still call Establish_Rpc_Receiver when the elaboration is completed.)

4.4 Partition ID

Type Partition_Id is an integer type defined in System.RPC. A value of this type is used to identify and access a partition. Do_RPC uses a value of this type as its first parameter to locate the RCI server that will execute the remote subprogram.

This implies that partitions have unique values: i.e., the location of a partition can be determined from only its partition id value. This raises the problem of making partition ids unique throughout the distributed system. For this we have at least two schemes:

- A specific partition server maintains a counter. The counter is a logical number that contains no implicit information. The matching between the logical number and a physical partition location (e.g., process number and port number) can be computed through a table look-up; this is a common situation for distributed systems.

- External facilities are used requiring integration of the port number and the process number to create a partition_id value. This information is sufficient if the same OS and the same data representation are used throughout the distributed system; however, this presents a problem in the context of heterogeneous systems. For example, in PVM [1], such identifiers (TIDs) are intended to be opaque to the application; thereby, disallowing attempts to modify or predict their values by the application. This is consistent with the objectives that lead to the design of the System.RPC interface. Another potential drawback with this approach is that it requires arbitrary long integers that would compromise portability.

4.5 Implementation Dependencies

System.RPC is the required minimal interface to the PCS. It offers the operations to marshal and unmarshal the subprogram parameters and to execute the remote call. Nevertheless, this interface is not complete and extensions are required. We indicate additional facilities that we need to implement the annex based upon the experience we have gained.

- Numbering the partition. In the previous section, we have outlined schemes for identifying partitions. In the first scheme a partition server is required together with elaboration code of a partition that requests a number. In the second scheme, we require explicit calls to the operating system to construct a composite value of the partition location. Therefore, both schemes have a dependency that prevents interchanging PCSs.

- Localizing an RCI server. This operation makes possible for a client to determine where the server it needs is. We have at least two approaches: a static and a dynamic one. The static one involves the replacement of the server reference by its physical address as soon as this information is known (at configuration time). Whereas the dynamic one requires the help of a server. In both cases, we have the notion of server information and there is a system dependency to fetch this information or to modify it.

- Establishing a correspondence between a partition number and a network address. In the previous section, we have mentioned two approaches to build partition id in a distributed system. These partition ids appear as logical values and the PCS needs to interpret them to reach the server with physical information (e.g., a network address, a port number and a processes number). We also have here a system dependency.

The above considerations indicate that there exist system dependencies in our implementation but also in any implementation. This is due in part to the limited interface given by System.RPC. When dealing with partition numbers, we need specific services to build a partition id and to interpret it and this is linked to the fact that there is no constructor-selector for type partition id: it is just an integer type.

We consider it as an illusion to build a universal PCS and in this respect the use of System.RPC might appear of limited interest. This interface appears even more useless as the programmer is not supposed to access it. We can find interests and benefits of System.RPC for the future. For example, in CORBA we have the notion of dynamically built requests to an object. With System.RPC, the system programmer is guaranteed to find the same interface on any implementation and then its extension is portable.

5 The system Architecture

5.1 The partition server

We have already presented the partition server in figure 1. We have selected this name in order to avoid confusion with what we can find in other systems. We can compare it to the portmap (SUN RPC) or to the Cell Directory Service of DCE. This database forms a kind of a clearinghouse where all the information on RCI servers is stored. The data can be retrieved with the key which is the name of the RCI package.

This server is centralized and it is known by all the partitions. Nevertheless, a client contacts the partition server only once. The information that is obtained from the partition server is stored locally and we have here a notion which is similar to the delegate of DCE.

5.2 The agent task on the server side

We maintain a pool of anonymous tasks that are responsible for RPC-receiver execution as depicted in figure 4. They receive a request from their environment and they execute a service. They also receive a handle so that the client is

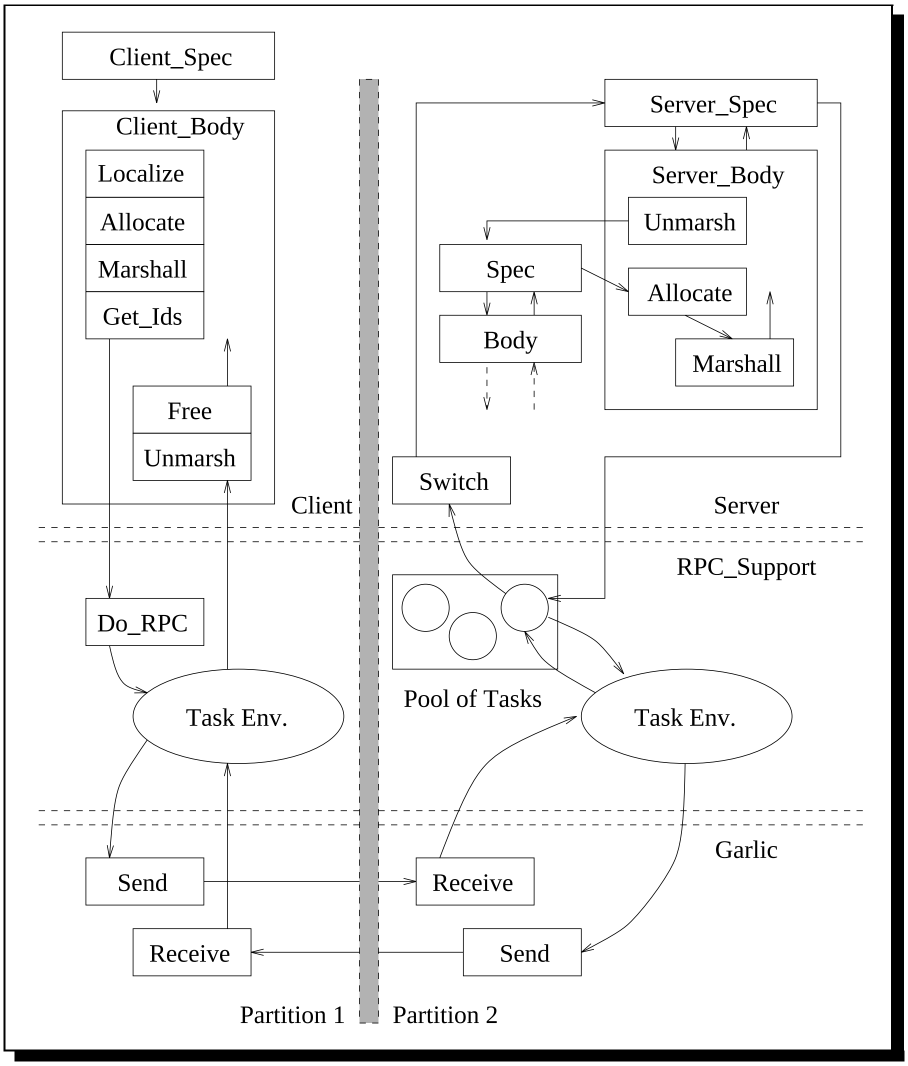

Figure 4: Our global architecture

reactivated at the end of the RPC. The number of these tasks is defined by our implementation and it can be increased to fit the needs of parallelism for some applications.

5.3 The environment task

This task is present on each partition and is waiting for incoming messages from the network. On the reception of a message, the environment task extracts the header of the message. We have two possibilities here:

- the header indicates that the message is a request for a remote service. The environment task selects an anonymous task in the pool, keeps track of it since the execution of the service might be cancelled by the client.
- the header indicates that the message carries the results of a RPC. This message is sent by the server which has completed the remote execution and now the client has to be reactivated. The header contains information so that the environment task knows which client is to be reactivated. The results of the RPC are then placed in the context of the client.

5.4 Garlic and communication issues

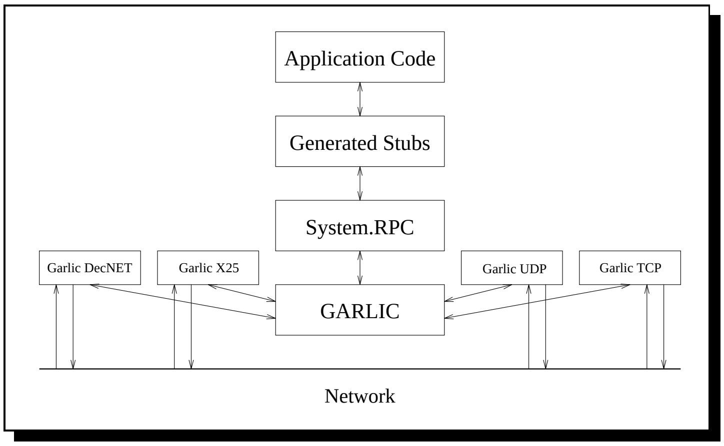

Figure 5: GARLIC and the network protocols

The GARLIC system can be considered as a medium-level communication protocol since it is used by System.RPC to exchange data between packages but it does not deal directly with communication channels (see figure 5). Low-level transport agents are implemented in child packages and thus can be changed at any time without need of recompiling GARLIC (although relinking may be necessary) and user transport protocols may be added at any time by derivation of the Protocol_Type tagged type A child package implementing TCP communications is provided under the name System.RPC.GARLIC.TCP.

System.RPC.GARLIC has no active thread of control of its own. Its services are either called by the upper layer (e.g., by the application level through System.RPC) when the application uses a communication channel or by a lower level (e.g., System.RPC.GARLIC.TCP acts an interface to TCP/IP) to inform GARLIC that data are available and some action is required to handle it.

GARLIC is built around a Synchronizer that is a protected object in Ada 95. This synchronizer ensures that concurrent accesses from the upper or lower layers will not corrupt its internal data and tables. Moreover, the entries implement event synchronization in a rather efficient way.

In a separate document [10], we detail the internals of GARLIC and its implementation and integration into the GNAT system. In the current version, GARLIC is made responsible to initialize the distributed application, to allocate unique identities to the partitions, to open the data connections between the partitions and also to transfer the data. In the future, we plan to extend GARLIC and to incorporate in it new services : deadlock and termination detection, clock synchronization, fault tolerance and reconfiguration support.

6 Implementation Experience

6.1 Target Systems

Network programming depends on the architecture and on the underlying operating system. They constitute critical points in any implementation as they are tedious and also very low level. That is the reason why we have isolated them in the structure of the PCS. Moreover by selecting PARADISE as the low level communication and POSIX-compliant interfaces, we benefit from several implementations of PARADISE on a wide range of platforms.

6.2 Work in progress and planned extensions

For the last few years, we have been working on environments for distributed system programming. Our major target was the design and the implementation of a repository of software reusable components. These components address and implement distributed control algorithms and programmers can tailor them to their needs [7]. In that context, we have been developing components for mutual exclusion, deadlock and termination detection, message-passing for real time applications and the Linda paradigm. We consider reusability as a major issue as distributed applications are complex and modularity is a way to manage their complexity.

Our current implementation targets immediate result and we have selected the most critical elements to be implemented as a demonstration of the annex. That is the reason we have not yet addressed the shared variables in the context of GNAT. The shared memory paradigm presents several benefits even in a distributed system where we have the notion of virtual distributed memory. In [8], we have proposed a new algorithm which implements such a memory and which is suitable for embedded applications. One of our aims is to incorporate this algorithm inside the GNAT system.

Fault tolerance is another concern and we have already integrated in GNAT the recovery block mechanism. This backward error recovery support has been implemented only in the context of sequential programs and we believe that its extension to distributed system would be of interest for the programmer.

The annex leaves any implementor free to add other paradigm. The RPC is of high interest but other paradigms (like Linda [4], Orca [2]) may fulfil the programmer's specific needs. Our point here is to offer several paradigms and to incorporate them in GNAT so that the programmer can select the most appropriate one to its requirements.

Another service of interest for the programmer is to offer type operations in the distributed systems.

6.3 Conclusions

Our implementation forms the basic core for distributed system programming. We have started to investigate fault tolerance issues, reconfiguration and virtual shared memory but numerous subjects can also be investigated by other individuals and teams.

7 Acknowledgements

The authors are very grateful to the GNAT team at New York University and specially to Prof. E. Schonberg and Prof. R. Dewar. L. Nana has implemented the stub generation in the GNAT compiler. The authors are also grateful to Prof. R. Volz from Texas A&M University who leads and manages the ADEPT project.

References

[1] Al Geist et al. *PVM : Parallel virtual Machine*. The MIT Press, 1994.

[2] H. Bal and A. Tannenbaum. Orca : a language for distributed object-based programming. Technical Report IR 140, Vrije University, Amsterdam, December 1987.

[3] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, 1984.

[4] N. Carriero and D. Gelernter. *How to write parallel programs : a first course*. The MIT Press, Cambridge, Massachussetts, 1991.

[5] DEC, HP, and et al. The common object request broker : architecture and specification. Technical Report OMG 91-12-1, Object Management Group and X Open, December 1991.

[6] E. Schonberg et al. GNAT: the GNU - NYU Ada translator, a compiler for everyone. In *Proceedings of the TRI Ada conference*, Baltimore, Maryland, november 1994. ACM.

[7] Y. Kermarrec and L. Pautet. Ada reusable software components for education in distributed systems and applications. In J.L. Diaz-Herrera, editor, *Proceedings of the 7th SEI conference on Software Engineering Education*, number 750 in Lectures Notes in Computer Science, pages 77–96, San Antonio, Texas, 1994. ACM IEEE, Springer Verlag.

[8] Y. Kermarrec and L. Pautet. Integrating page replacement in a distributed shared virtual memory. In *Proceedings of the 14th international conference on distributed computing systems*, Poznan, Poland, June 1994. IEEE.

[9] Y. Kermarrec, L. Pautet, and E. Schonberg. Design document for the implementation of distributed system annex of ada 9x in gnat. Technical report, New York University, Courant Institute, 715 Broadway, New York NY 10012, March 1995. (to be published).

[10] Y. Kermarrec, L. Pautet, and S. Tardieu. Garlic : Generic ada reusable library for interpartitions communication. Technical report, ENST de Bretagne, March 1995. (soumis).

[11] Yvon Kermarrec and Laurent Pautet. Evaluation of the distributed systems annex of Ada 9X and its implementation in GNAT. Technical report, Courant Institute of Mathematical Sciences, New York University, 715 Broadway, New York NY 10012, 1994. to be published.

[12] Message Passing Interface Forum. Mpi : a message passing interface standard. Technical Report 230, CS Department, University of Tenessee, Knoxville, April 1994.

[13] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE.* O'Reilly and associates, inc, 1993.