Building Modern Distributed Systems

Laurent Pautet, Thomas Quinot, and Samuel Tardieu

École Nationale Supérieure des Télécommunications
Networks and Computer Science Department
46, rue Barrault
F-75634 Paris Cedex 13, France
{pautet,quinot,tardieu}@enst.fr
http://www.infres.enst.fr/

Abstract. Ada 95 has been the first standardized language to include distribution in the core language itself. However, the set of features required by the Distributed Systems Annex of the Reference Manual is very limited and does not take in account advanced needs such as fault tolerance, code migration or persistent distributed storage.
This article describes how we have extended the basic model without abandonning the compatibility in GLADE, our implementation of the Distributed Systems Annex. Extensions include restart on failure, easy code migration, hot code upgrade, restricted run time for use on embedded systems with limited processing as well as distributed storage capabilities and persistent storage handling.

1 Introduction

It is generally admitted that Ada 83 had a strong focus on real-time, mission-critical systems. But Ada 83 has been criticized from a number of standpoints, one of them being its lack of cooperation with foreign programming languages and with the outside world in general. To fix those defects, new features were added in the latest major revision of Ada, called Ada 95 [1]. Moreover, Ada 95 was the first internationally standardized OO language (ANSI/ISO/IEC-8652:1995). It is also the first internationally standardized language including distribution features.

A great effort was led by Ada Core Technologies (ACT) to provide the Ada community with a free high-quality Ada 95 compiler called GNAT. This compiler, which implements the core Ada 95 language as well as all its optional annexes, belongs to the GCC family and shares its back-end with the C and C++ compilers. As with the other compilers from the GCC suite, GNAT supports many native and cross configurations.

In collaboration with ACT, we have been developing GLADE [2], an implementation of the Distributed Systems Annex of the language, as found in the Ada 95 reference manual. GLADE, which is available under the same free license as GNAT, has been designed for this particular compiler, but should be portable to any Ada 95 compilation environment with minimal efforts. We also worked on proposing new extensions to the original Ada 95 model for distributing Ada programs, which is described in section 2.

Those extensions, whose goal is to allow the use of modern distribution paradigms while maintaining total compatibility with the basis model describe in the Distributed

Systems Annex, have been inspired by other middlewares such as CORBA [3] or by user needs and remarks. The first extension, presented in section 3, removes the single point of failure often found in distributed systems. Section 4 focuses on restarting parts of a distributed systems after a failure, which may have been scheduled (in case of code migration or upgrade) or not. Section 5 concentrates on shared and persistent data storage.

In section 6, we present other useful features such as encryption or data compression. We then conclude and present our current and future research work in the last section.

2 Distribution in Ada

In this section we present the distribution model of Ada 95, then analyze its intrinsic limitations as well as the ones present in older GLADE releases.

2.1 Distribution Model

Ada 83 lacked distribution facilities. Every compiler vendor had to provide its own proprietary solution for letting users build distributed applications. This led to situations where a program could not easily be ported to another compiler[1], which went against Ada's implicit rule of "as soon as it is written in Ada, it will work the same way with every conforming Ada compiler". In fact, there were so many different solutions that a comparative study had already been made in 1985 [4].

The designers of Ada 95 chose to solve this problem by adding distribution features right in the language. This led to the "Distributed Systems Annex" of the Reference Manual [1, Annex E]. This annex describes how an application can be split into different **partitions** (as described in [5]). Partitions can be **active**, in which case they can contain threads of control and packages with code, or **passive**, meaning that they only hold passive packages, containing variables. The annex also defines how particular packages belonging to the application can be categorized using **categorization pragmas**. Those pragmas identify the packages playing a special role in a distributed application; they come in addition to regular categorization pragmas such as Pure and Preelaborate. The additional pragmas are:

Remote_Call_Interface: subprograms declared in a Remote_Call_Interface package will not be replicated on all the partitions where they are used. Each such package is placed on only one active partition in the distributed application. When calling a subprogram declared in a Remote_Call_Interface package, a remote call takes place transparently if the package has been placed on a partition different from the one of the caller.

Remote_Types: types declared in a Remote_Types package are guaranteed to be transferable from one partition to another. Notably, those types have global semantics; for example, pointers are good examples of type whose semantics are only local,

[1] To be honest, this problem could have been solved in an elegant way if Ada 83 had a standardized way of interfacing with other languages, which was not the case.

since it makes no sense to transport them on another machine. At the opposite, integers have the same meaning on every partition.

Shared_Passive: variables declared in a Shared_Passive package can be accessed from several partitions. Simple variable assignments and reads can be used to exchange data between partitions. Shared_Passive packages can be placed on either active or passive partitions.

As soon as a package holds the Remote_Call_Interface pragma, the subprograms present in its declarative part can be called remotely. When necessary, the compiler will generate stubs and skeletons to make a remote call; categorization dependency rules guarantee that this will be doable.

This model allows to switch from the monolothic model to the distributed one (and vice-versa) very easily, thus easing the debugging of distributed applications. However, the Reference Manual does not say anything about the way a distributed application gets built, just as it does not describe the compiler comand line options in the case of a regular program. In GLADE, we chose to create an external tool called GNATDIST, described in [6] and [2]. This tool takes a configuration file written in an Ada-like language and produces one executable per active partition; it shares code with GNATMAKE, GNAT's tool for building non-distributed programs.

GNATDIST allows the designer of a distributed application to apply pragmas and attributes on partitions. Using them, it is possible to set properties on partitions, such as their behaviour when a service becomes temporarily unavailable, as is described in section 4.

2.2 Limitations

The Ada 95 model for distributing applications has been a big win over Ada 83, but is still very limited with regards to today's needs. We strongly feel that important features and network protocols should also have been standardized, to increase Ada 95 interoperability. To take an example, a validated implementation does not have to support heterogeneous systems, when Intel-based and SPARC-based machines are involved in the same distributed program[2]. Also, an implementation made by a compiler vendor will probably not be compatible with an implementation made by another vendor.

Also, the behaviour of a distributed application in case of a partition's failure is undefined. This is consistent with the non-distributed model conception, where the disappearance of a part of the code from memory is not taken in account; it just cannot happen on a working hardware with a working operating system.

More advanced concepts such as safety and integrity of the communication have not been integrated in the annex either. It is generally assumed that the network and the computers are under complete control, and that no attacker can snoop or alter data packets (our solution for precisely this limitation can be found in section **??**).

Former versions of GLADE also had limitations. For example, some versions had a single point of failure, called the **boot server**; it was the main partition of a distributed application, and the only entry point for new partitions. Its disappearance made

[2] Note that GLADE fully supports heterogeneous systems.

it impossible to add new partitions to the running distributed program. As explained in sections 3 and 4, this limitation has been removed in the last version of GLADE. Also, another limitation that has been recently reduced and will be detailed in section 6 was that every partition contained the whole run time; this was a real problem in embedded systems where memory and processing power are both expensive and constrained.

3 Removing the Single Point of Failure

As we have seen in section 2.2, the boot server is the weakest part of the distributed application; if it dies, no more partition will be able to join the running distributed system. For this reason, we have added to GLADE the capability of having more than one boot server, using what we call **boot mirrors**.

A ring of boot mirrors connected together act as a replicated boot server. Tree-like structures could have been used instead of a ring; we chose a ring because it was easier to reconfigure in case of a partition failure.

A partition wishing to join a running distributed application needs to know the address of one of the boot mirrors and connects to it.The boot mirror will then propagate data about the new partition to the whole distributed program, and it will also provide the new partition with all the needed information to become an integral part of the distributed system.

Note that the presence of boot mirrors does not obsolete the boot server: one of the boot mirrors is considered to be the boot server. Its role is to launch the global wave-based termination detection algorithm similar to the one found in [7] and refined in [2].

While a single boot server can take all the decisions such as assigning a Partition_ID[3] for the new partition by itself, the various boot mirrors have to negotiate to avoid possible race conditions. One potential problem is if two instances of the same partition try to connect simultaneously to two different boot mirrors. Each of them will check that the partition is not already present in the distributed system, which will be the case. However, only one of the two partitions must be accepted, and the other one must be disconnected soon enough not to have received any request from third-party partitions. To solve this issue, different startup algorithms are used in this boot mirrors ring. One of them handles Partition_ID allocation, and another one with the Remote_Call_Interface package declaration and version check. The interested reader will find descriptions of those algorithms using the high-level Petri nets formalism in [2].

The boot mirrors ring can reconfigure itself dynamically, and can be extended or shrunken. When a new boot mirror wants to join the distributed application, it connects as any other partition does by contacting an existing boot mirror. Once it has been fully added to the running distributed application, it can insert itself in the ring and from there act as a boot mirror. If a boot mirror disappears, its predecessor and successor connect to each other and form a shrunken ring. All the partitions who previously chose the now-dead boot mirror contact another one from the last list of boot mirrors they got. Also, a new boot server is elected among the set of existing boot mirrors.

[3] A Partition_ID is an integer used to identify a partition in a running distributed system [1, E.1(8)].

To prevent early death of the only boot mirror, a new attribute **Mirror_Expected** has been introduced in GNATDIST. Its presence prevents the distributed application from starting remote procedure calls until at least two boot mirrors are present, so that one of them can die without compromising the liveness of the whole distributed application.

However, when a partition offering an active service (such as through a Remote_Call_Interface package) dies, it may make the whole distributed application useless. The next section shows how a dead partition can be restarted or upgraded dynamically.

4 Fault Tolerance and Code Migration

There are several reasons why a partition must be restarted:

1. A failure occurred and the machine on which the partition was running is no longer reachable.
2. The administrator of the distributed systems decided that the machine on which the partition was running could not afford the load anymore; the partition is then stopped and must be restarted on a new node (this is also called **code migration**).
3. Errors or inefficiencies have been detected in the implementation of the services offered by the partition, or new services have been implemented. The old code will be replaced by the new one (**hot code swapping**).

In any case, the distributed application as a whole must be notified that a service has become unusable; decisions must be taken regarding the behaviour to adopt when this situation arises, depending on whether the service is supposed to be restarted or not, and whether the service is strictly necessary for the good health of the distributed program.

In section 4.1, we show how a partition can be restarted. Section 4.2 describes the various behaviours that can be adopted by the other partitions of the distributed application.

4.1 Restarting a Partition

The Distributed Systems Annex does not say anything about the name of a partition. A partition is known only through its Partition_ID, which must be unique in the distributed application at any time. This Partition_ID is obtained through the attribute of the same name applied to packages or subprograms, and it can only be used for comparaison with other such attributes. However, nothing in the Reference Manual forces the value of this attribute to stay the same during the whole life of the distributed program.

This facility has been used in GLADE to implement service restarting. When a partition offering a service becomes unavailable, its Partition_ID will not be reused at any later time. However, this service can be restarted on a new partition, which will get its own new Partition_ID.

The only constraint put on a service being restarted is to keep the same Version, that is to have the same declarative part as the one it replaces. It can be restarted on another machine with another architecture, or have a totally different implementation. Two direct consequences are that code migration and hot code swapping can be achieved very easily by using the right reconnection policy, as described in the next section. State data about the service can be kept using methods described in section 5.

4.2 Failure Handling Policy

A client using a service can adopt different behaviours when the service goes away. We have implemented three different behaviours, chosen through the Reconnection attribute of GNATDIST:

Reject_On_Restart : this policy is the strictest one. Once a service has been started, there is no way it can be replaced if it dies. This ensures that no inconsistency can be introduce by loss of data or incomplete transactions. Any call to such a service after its death will raise Communication_Error, even if it tries to start again.

Wait_Until_Restart : this policy makes any attempt to call the remote service blocking until the service has been revived. From the client point of view, no call to the service will ever fail, except when a call is in progress while the service is being disconnected. However, there is no guarantee that the client will not hang forever.

Fail_Until_Restart : this policy is a compromise between the two others. While the service is absent, Communication_Error will be raised. When it comes back, clients will be served again as if nothing happened. This allows a client to use a service when possible, or to use a fallback one if the main service is unavailable without blocking forever.

We have been recently proposed a fourth failure handling policy, whose descriptive name could be Wait_Until_Restart_Or_Timeout. We prefer the use of the "select ... then abort ... end select" construct in conjunction with the Wait_Until_Restart policy to get the same result.

5 Preserving Partition State

It is useful to be able to revive a dead partition. It is even better if the partition can restart with a meaningful consistent state. One of the ways to achieve this is to preserve the state of the partition at some specific points on a persistent data store to be able to restore it later.

Preserving the state of a partition can be performed at different programming levels. For example, the user can manually save all the pertinent partition data on a persistent storage object such as a file-system. This is error-prone as no automatic mechanism can ensure that the whole state has been saved.

A more transparent solution consists in using Shared_Passive packages. These categorized packages contain the declaration of shared variables: global data can be shared between active partitions, providing a repository similar to a shared memory, a shared file-system or a database. Entry-less protected objects offer safe concurrent access and update of shared objects. This feature is orthogonal to the notion of distributed objects, which are only accessed through their exported methods. Shared_Passive packages can be configured on both active and passive partitions. An active partition comprises one or more threads of control, whereas a passive partition must be pre-elaborated and may not perform any action that requires run time execution[4]. Typically, a passive partition can be seen as a global address space shared by several active partitions.

[4] In fact, those limitations only apply to user code and a compiler is free to add any code deemed necessary to perform the expected operations. However, it is against the spirit of passive par-

5.1 Shared_Passive Packages Implementation

In the GLADE/GNAT model, each partition that includes a Shared_Passive package has its own local copy of the package data. This local copy can have an initial state if the data storage used for this partition is persistent. In GNAT's implementation, this property is achieved by maintaining a set of files, in a dedicated directory. GLADE's implementation provides this particular storage along with additional ones.

Each variable v from a Shared_Passive package p gets its own file named after the fully qualified name of the variable, here "$p.v$". When a partition needs to read the value of variable v, it checks for the existence of this file. If it does not exist, the in-memory value of v is used, which corresponds to the initial value (if any) given at variable declaration time. If the file exists, the value stored in the file is used. Assigning a new value to v will create or update the content of the file. Therefore, this model automatically provides persistence assuming the underlying storage support lifetime is longer than the one of the program execution. It is up to the persistent storage interface to choose when the data is really committed to the persistent store. An easy choice could be at "Shared_Var_Close" time.

GNAT Implementation Issues For each shared variable v of type T, a read operation "vR" is created whose body is given in figure 1. The function Shared_Var_ROpen in package System.Shared_Storage either returns null if the storage does not exist, or returns a Stream_Access value that references the corresponding shared storage in which the current value will be read.

```
procedure vR is
   S : Ada.Streams.Stream_IO.Stream_Access;
begin
   S := Shared_Var_ROpen ("p.v");
   if S /= null then
      T'Read (S, v);
      Shared_Var_Close (S);
   end if;
end vR;
```

Sample 1: Read expansion

Each read operation of v is preceeded by a call to the corresponding "vR" procedure, which either leaves the initial value unchanged if the storage does not exist, or reads the current value from the shared storage if it does. In addition, for each shared variable v, an assignment procedure is created whose body is given in figure 2. The function Shared_Var_WOpen in package System.Shared_Storage returns a Stream_Access value that references the corresponding shared storage, ready to write the new value.

titions to embed code in them, as they should be placeable on strictly passive nodes such as a pure memory area.

```
procedure vA is
   S : Ada.Streams.Stream_IO.Stream_Access;
begin
   S := Shared_Var_WOpen ("p.v");
   T'Write (S, v);
   Shared_Var_Close (S);
end vA;
```

Sample 2: Assignment expansion

Each assignment operation to v is followed by a call to the corresponding "vA" procedure, which writes the new value to the shared storage.

The call to procedure Shared_Var_Close indicates the end of a read or assignment operation. When a read operation and an assignment operation occur at the same time on the same partition, as the same stream is used simultaneously, both operations can terminate abruptly by raising an exception. Such a fatal error may occur when the stream has been opened in read mode (call to "vR") and then in write mode (call to "vA") and at least used by the read operation (call to T'Read). To avoid this unfriendly behaviour, we introduced an additional mutual exclusion at the partition level. This GNAT expansion always takes place, whether the user works in the distributed environment of GLADE or in the non-distributed environment of GNAT.

GLADE Implementation Issues GLADE provides a data representation based on XDR [8]. As GNAT's expansion is based on streams, heterogeneity is not a problem even for Shared_Passive packages shared between partitions running on different architectures.

Like a Remote_Call_Interface package, a Shared_Passive package has to be unique in the overall distributed system [1, E.2.1(10) and E.2.3(17)]. Moreover, a version check has to be performed to ensure that the package specification used at execution time is consistent with the one used at compilation time [1, E.3(6)]. In the GLADE environment, a Shared_Passive package like a Remote_Call_Interface package has to register to the boot server during its elaboration code in order to declare its partition location.

For these reasons, GLADE generates specific elaboration code for the client stubs and the server skeleton. The server skeleton of a Shared_Passive package P registers information about itself onto the boot server, and then checks that it has been correctly registered by performing a request concerning itself. The client stubs retrieve information about the package and check the result against the information concerning the package specification as known at compile time.

5.2 Passive Partition Implementation

An interesting problem is raised by passive partitions, because they are not able to perform any action at run time, as they have no thread of control of their own. Therefore, active partitions that have visibility on Shared_Passive packages configured on passive partitions have to act in place of those partitions. Multiple registrations problems are solved by requiring that each passive partition in a distributed program has a unique name. For the same reason, Shared_Passive packages configured on such a partition are

registered by the active partitions. The first registration of a Shared_Passive package is assumed to be authoritative; any further registration will be checked for consistency against this first registration.

5.3 Various Shared Storage Support

GLADE has a modular, layered and object-oriented architecture [9] which makes it easy to add new communication protocols. The important modules in this context are the core of GARLIC, called *Heart*, and the protocols. The protocol layers know nothing concerning the format of the data they convey, and the high-level layers will work on any protocol. All protocols inherit from a common abstract protocol class. To implement a new protocol, the developer overrides abstract methods of the base protocol class.

GLADE uses the same architecture for storages; every storage inherits from a common abstract storage class, whose methods will be redefined. Existing storage supports include GNAT's file-systems support, but also two other ones that have been recently added. One of them is based on a distributed shared memory algorithm, and the other one uses a fault-tolerant distributed database manager.

The user can configure the Shared_Passive packages and passive partitions by using GNATDIST. To support the configuration of passive partitions and storage supports, GNATDIST introduces two new attributes **Passive** and **Data_Location**.

The Passive attribute must be applied to a partition to indicate that this partition is passive. GNATDIST checks that it only holds Shared_Passive packages. GNATDIST allows to configure the network location of an active partition through the **Self_Location** attribute. This location contains the protocols with their internal data to use to communicate with this partition. It is also possible to configure the storage supports with their internal data to use to get access to shared objects from Shared_Passive packages configured on an active or passive partition through the **Data_Location** attribute. For instance, a developer using the GNATDIST configuration language could write the following representation clause.

for Partition'Data_Location **use** *"dfs://dir"*

This clause configures all the partitions storage supports to "dfs" which stands for Distributed File System; the directory used by the underlying storage support (probably NFS) is "dir".

5.4 Distributed File-System Storage

The basic storage support is based on a distributed file-system storage support. To safely share files among several partitions, the user must ensure that the partitions that reference shared objects have access to an operating system service such as NFS [10]. Also, some distributed file systems do not allow that two processes open the same file for writing at the same time; this can cause priority inversion problems, as various tasks with different priorities may be unblocked at the OS discretion.

5.5 Distributed Shared Memory Storage

This storage support provides an implementation of a distributed memory based on the well-known algorithm of Li and Hudak [11] [5].

Many algorithms have been proposed to maintain a strong memory coherence in a distributed shared memory. In the most basic algorithm, an object server is present on every partition of the distributed system and the servers are in charge of maintaining the consistency of the distributed shared memory. When a partition wants to access an object, it contacts its local server and two situations can occur: the object is available or it is not. If the object is unavailable, the server makes a read/write object fault in order to get the object from the others.

In a first approach, a server devoted to an object centralizes write and read operations. It receives these requests, executes them and sends acknowledgements or object copies. Naturally, the object server (or the object owner) may be overloaded by too many write requests especially when the object locality is not adequate. Thus, another strategy allows the object to migrate to the last client which becomes the new object owner. Read request bottleneck has a more flexible answer since the uniqueness of the object in read-mode is not required. Therefore, object replicas may be delivered by the object owner to several clients as long as the object is not modified. An invalidation protocol ensures that any write operation invalidates all object replicas.

GLADE's distributed shared memory is based on this algorithm with object migration and read-only replicas. This algorithm is very efficient in terms of network activity but does not provide fault-tolerance properties. Therefore, we plan to implement another algorithm for distributed memory which provides full object replicas [13].

5.6 Fault-Tolerant Database Storage

We have implemented one more back-end for Shared_Passive package, based on an existing soft real-time fault-tolerant distributed database manager, called Mnesia. Mnesia is written in Erlang[6], a language used primarily for building telecommunication switches [14].

Just as Ada, Erlang integrates distribution features right in the language. More exactly, the Erlang model is based on inter-process communication, but without any consideration about the physical location of the target process, which can be located on another Erlang node. The Mnesia database system allows an Erlang application to store any term and retrieve it with a $O(1)$ complexity in a read-only table[7]. Data can be accessed transparently from any Erlang node, and can be replicated on one or more nodes and with a reasonable complexity in more complicated cases.

We wrote this back-end with the assumption that a crash in an Ada distributed application was due to a network or a hardware failure, not to a fault in the application. Also, Erlang nodes are robust and not likely to crash, as the Erlang virtual machine takes care of all memory allocations and deallocations without letting the user manipulate pointers

[5] The study of an Ada implementation of this algorithm can be found in [12].

[6] See `http://www.erlang.org/` for more information on Erlang.

[7] A read-only table does not need exclusive access, as opposed to a read-write one.

at all. Erlang even offers the programmer with automatic supervision, and can restart important threads if they die unexpectedly. Of course, an hardware failure will also affect an Erlang node, which is why Erlang keeps its databases replicated. We have then chosen to associate one local Erlang node to each Ada partition: an Ada partition P_i and its associatiated Erlang node E_i are located on the same machine.

Note that not all the Erlang nodes need to have a copy of the database holding the Shared_Passive state. However, to survive k simultaneous failures, at least $k+1$ replicas of the database must exist.

6 Other Useful Features

The loose requirements of the Distributed Systems Annex over the internal behaviour of the distribution run time allowed us to implement additional features while staying fully compatible with the Reference Manual. Some of those features have already been described in details in other articles; they will only be briefly summarized here for completeness.

Data Filtering: Incoming and outgoing data can go through a user-defined filter in order to provide services such as encryption or authentication [15];

Termination Policies: GLADE extends the classical termination model and allows for example clients to terminate while servers keep running [2];

Light Run-Time: in some particular configurations, GLADE can detect that a partition does not need to embed the whole distribution run-time. For example, if it can decide that a partition has a single-threaded client-only behaviour, then it can choose to include a light run-time that will not use any tasking.

7 Conclusion and Future Work

In this paper, we have shown how fault tolerance, code migration and data persistence have been added without giving up the compatibility with the language concepts described in the Reference Manual.

We are currently pursuing our research work in two directions:

1. We are extending the list of platforms that GLADE supports; support for JGNAT [16] is on its way, and will support distributed applications made of native and bytecode partitions.
2. We are working on bridges between the Distributed Systems Annex and other middleware, such as CORBA. We have already released ADABROKER[8], a free software CORBA implementation written in Ada. Our goal is to eventually use a common network layer and communication stack in both ADABROKER and GLADE. This could lead to the choice of IIOP, CORBA's standardized communication protocol for the Internet, as the underlying GLADE protocol.

Our goal is to continue to extend the range of domains that can be reached by Ada 95 distributed systems as much as possible.

[8] ADABROKER is available at `http://adabroker.eu.org/`.

References

1. ISO, *Information Technology – Programming Languages – Ada*. ISO, Feb. 1995. ISO/IEC/ANSI 8652:1995.
2. S. Tardieu, *GLADE – Une implémentation de l'annexe des systèmes répartis d'Ada 95*. PhD thesis, École Nationale Supérieure des Télécommunications, Oct. 1999. PhD advisor was L. Pautet.
3. L. Pautet, T. Quinot, and S. Tardieu, "CORBA & DSA: Divorce or Marriage?," in *Proceedings of AdaEurope'99*, (Santander, Spain), June 1999.
4. J. W. Armitage and J. V. Chelini, "Ada software on distributed targets: a survey of approaches," *ACM SIGADA Ada Letters*, vol. 4, pp. 32–37, Jan./Feb. 1985.
5. A. Gargaro, S. J. Goldsack, C. Goldthorpe, D. Ostermiller, P. Rogers, and R. A. Volz, "Towards distributed systems in Ada 9X," in *Proceedings of the Conference for Industry, Academia and Government*, (New York, NY, USA), pp. 49–54, ACM Press, Nov. 1992.
6. Y. Kermarrec, L. Nana, and L. Pautet, "GNATDIST: a configuration language for distributed Ada 95 applications," in *Proceedings of Tri-Ada'96*, (Philadelphia, Pennsylvania, USA), 1996.
7. F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, no. 3, pp. 161–175, 1987.
8. Sun Microsystems, *xdr – library routines for external data representation*. Unix systems manual page.
9. Y. Kermarrec, L. Pautet, and S. Tardieu, "GARLIC: Generic Ada Reusable Library for Interpartition Communication," in *Proceedings Tri-Ada'95*, (Anaheim, California, USA), ACM, 1995.
10. J. Corbin, *The Network File System For System Administrators*. Mountain View, Californie, USA: Sun Microsystems, Inc., 1993.
11. K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, November 1989.
12. Y. Kermarrec and L. Pautet, "A Distributed Shared Virtual Memory for Ada83 and Ada9X Applications," in *Proceedings of TriAda'93*, (Seattle, Washington, USA), Sept. 1993.
13. K.-L. Wu, K. Fuchs, and J. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 231–239, April 1990.
14. J. Armstrong, M. Williams, and R. Virding, *Concurrent Programming in Erlang*. Englewood Cliffs, NJ: Prentice-Hall, 1993.
15. L. Pautet and T. Wolf, "Transparent filtering of streams in GLADE," in *Proceedings of Tri-Ada'97*, (Saint-Louis, Missouri, USA), 1997.
16. C. Comar, G. Dismukes, and F. Gasperoni, "Targeting GNAT to the Java Virtual Machine," in *Proceedings of the TRI-Ada'97 Conference, November 9–13, 1997, St. Louis, MO* (ACM, ed.), (New York, NY 10036, USA), pp. 149–164, ACM Press, 1997.