Building fault tolerant distributed systems using IP multicast

Samuel Tardieu
sam@inf.enst.fr

Laurent Pautet
pautet@inf.enst.fr

École Nationale Supérieure des Télécommunications, Paris, France
Network and Computer Science department
http://www.inf.enst.fr/

Abstract

Our institute has been developing the only publicly available implementation of the Ada 95 Distributed Systems Annex for several years in strong collaboration with Ada Core Technologies. We are now extending this implementation far beyond the requirements of the Reference Manual by reducing the memory footprint and adding fault tolerance. This paper will present the progress made on the latter.

So far, we have been using IP multicast and forward error correcting techniques successfully for data broadcasting as well as distributed semaphores for synchronization, to implement a one-writer, multiple-readers shared memory area. This allowed us to suppress the only single point of failure that existed in our implementation. We plan to use this model as a framework for construction of fault tolerant distributed systems.

1 Introduction

1.1 Distributed systems

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

Processing throughput: if the program is computation intensive, then a single processor may not be powerful enough 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 hardware 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 transient network 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 may have an entry point (the main procedure) but may be separated into *partitions* in a semi-transparent way. 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. The programmer will use pragmas as described in annex E of the Reference Manual ([8]) to indicate where the program will be split, and an external tool will do the effective partitioning (see [7]).

1.2 GLADE, the current implementation

Right now, GNAT (see [18]) is the only compiler supporting the Distributed Systems Annex. To build a distributed application, GNAT must be used with GLADE, a freely available package[1] made of two main components:

GNATDIST is a tool that reads a configuration file (the configuration language is described in [9]) and builds all or part of the distributed program

GARLIC (described in [10]) is the run time library implementing network communications and advanced features such as transparent filtering (see [15]), encryption, remote abortion, package to partition mapping and partition localization

GLADE is being jointly developed by ENST and Ada Core Technologies[2] under the General Public License (GPL). A technical overview of GLADE can be found in [20] and [19].

1.3 Our project

The Network and Computer Science department of ENST is aiming at building a complex distributed system which involves Sparc workstations as well as mobile robots originally built for the SPIF project (see [4]). The robots' main CPU is a 68360 running the real-time OS RTEMS 4.0 (see [14]), for which GNAT is already available; the workstations are running Solaris 2.5.1.

[1] GLADE can be downloaded from `ftp://ftp.cs.nyu.edu/pub/glade/`.

[2] ACT is also selling support contract for GNAT and GLADE professional; see `http://www.gnat.com/` for more information.

The robots evolve in the real world and do not communicate with each other directly. All the communications go through the cluster of workstations and can eventually be forwarded to other robots if needed, as described on figure 1.

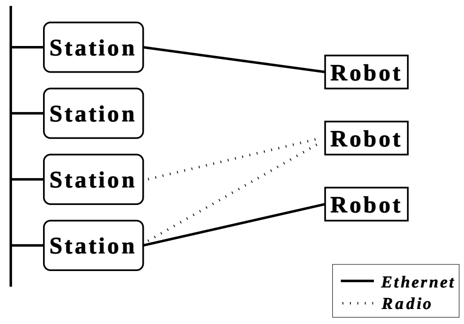

Figure 1: Mobile robots and workstations

Of course, one cannot imagine building such a complex system without introducing fault tolerance at one level or another; it would be unacceptable to shutdown the whole system just because one of the nodes has suddenly crashed. Also, we do not need to embed all the great and many features of GLADE in every robot, because doing this would mean adding a lot of RAM onto each entity (right now, each mobile robot has 4 megabytes of RAM which contains both code and data).

For this purpose, a light run time library has been implemented, which dramatically reduces the overhead usually introduced by the distributed run time environment. For example, this light run time system does not drag the whole tasking code into the executable if tasking is not used explicitly in the user's code. This part has been fully implemented and will constitute the main topic of an upcoming research paper.

The fault tolerance issue has been restricted to the cluster of workstations for obvious reasons: since robots are likely to be randomly located, it would make little sense to load data from one robot into another, for the latter will perceive a completely different environment. Also, we have no guarantee that efficient communication channels between robots and workstations will be available at any time, while we can at least suppose that the workstations will be connected to each other using a permanent, fast link.

Each robot is able to communicate with a new workstation if the one it was talking to comes down. In other words, from the robot's point of view, the cluster of workstations is always up and running.

2 The multicast technology

Despite several years of existence and heavy use in some fields of application, the IP multicast protocol is still unknown to many people. In this section, we will describe the basis of this technology, which we are using for our developments.

2.1 Unicast and multicast

Computers willing to communicate with each other have to use a common set of protocols to be able to exchange data. On the Internet, the IP protocols family (see [5]) is widely used, especially the TCP protocol which provides reliable FIFO communications and the UDP one which is non-reliable and does not guarantee packet ordering. These two protocols are called point-to-point protocols because they can only be used to exchange data between two computers identified by IP addresses and port numbers.

These protocols are unsuitable for large scale broadcasting such as video or audio conferencing. If a conversation involves n participants and if each participant must receive every piece of data sent by the others, $n.(n-1)$ channels must be established if each participant is directly connected to the others. It is possible to work with only n channels if there is a central hub acting as an exchange center, but it introduces an extra delay that can dramatically increase the round-trip time and thus reduce the audio quality as well as the frame rate of video sequences. The two schemes are shown on figure 2 (links between nodes are virtual links and do not correspond to physical links). We can clearly see that with these protocols, each packet sent by a European participant is sent twice on the (supposedly unique) transatlantic physical link.

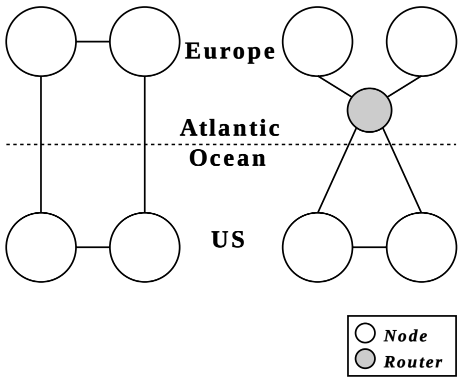

Figure 2: Point-to-point communication

To solve this problem, a new protocol called *IP multicast* has been introduced. The idea behind this protocol is to never propagate the same packet more than once on the same physical link. IP addresses are no longer used to designate participants; each participant subscribes to a *multicast channel*, and routers collaborate to propagate every packet to all the participants (see [6] for examples of applications using IP multicast). Figure 3 shows how the various routers communicate together and with the final hosts to provide data packets to all the participants.

The routing protocol used for multicast is an intelligent routing protocol: it will not propagate packets over a link if there are no subscribers of the multicast channel beyond this link. This protocol is central to our implementation of low bandwidth data replication because it ensures efficient communication between partitions, thus allowing to exchange a huge amount of data which could not have been propagated to all the partitions using the previously described point-to-point model because it would have required more bandwidth than available.

Even in complex cases such as the one shown in figure 4, the routing protocol will choose the best route depending on the load of each link.

The IP multicast protocol has been standardized by the Internet Engineering Tasking Force (IETF) as other Internet protocols and is described in [1].

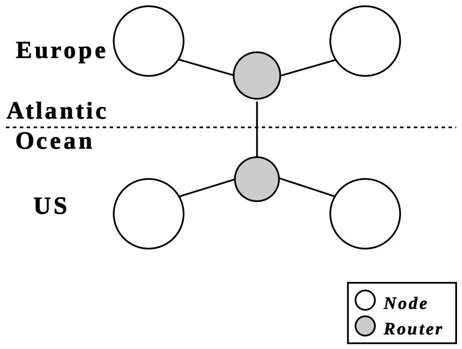

Figure 3: Multicast communication

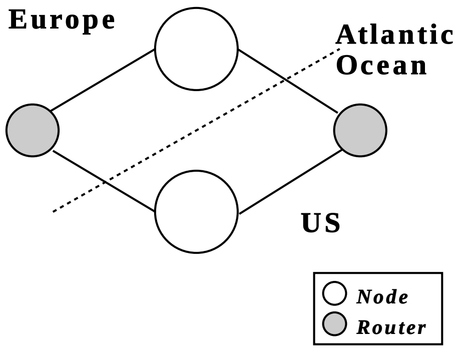

Figure 4: Diamond configuration

2.2 Forward error correcting

IP multicast being a non-reliable protocol, it may suffer random packet loss due to network congestion or insufficient processing throughput in routers or end-points. The *Forward Error Correcting* (FEC) technique consists in sending self-correcting redundant packets to reduce the probability of packet loss.

Let us assume we have n packets $P_1, P_2, ..., P_n$ of length L that we want to transmit using an unreliable protocol, and that the probability of loosing more than one packet out of $n+1$ is very low. We can build an extra packet P_{n+1} using an *exclusive or* function (represented as $\oplus$):

$$P_{n+1} = \bigoplus_{i=1}^{n} P_i$$

If a packet j gets lost during the transmission, it is possible to reconstruct it by computing:

$$P_j = \bigoplus_{i=1, i \neq j}^{n+1} P_i$$

If the packets do not all have the same length, the shortest ones are padded with zeroes. The length of the packet itself is being included in this algorithm, thus making it possible to rebuild packets of different lengths.

It is of course possible to transmit more than one redundancy packet; a polynomial function is then used to build those extra packets. A compromise has to be found between the total bandwidth used by FEC packets and the one used by extra retransmission if FEC is not (or insufficiently) used. Also, retransmitting a packet introduces an extra delay due to the packet round-trip time.

We have extended the FEC algorithm to dynamically change the number of FEC packets depending on the effective losses measured on the various links. While a request for retransmission and its response take at least two packets (if it is done using an unreliable protocol, it may have to be repeated, and if it is done with a reliable protocol, this protocol will automatically include extra control packets to make sure that the data arrived to the target), it is sometimes more efficient not to adapt the algorithm to the worse link because every communication link would get the extra FEC packet. The number of FEC packets depends on the average loss of all the links and on the number of links whose loss is above this value.

2.3 Experiment reports

We have measured the effect of the FEC algorithm on the bandwidth in several situations. Figure 5 shows a simulation, where 10 packets have been completed by 0, 1, 5 or 10 FEC packets.

We see on this figure that for very good links (loss close to 0%), the use of FEC will increase the bandwidth and will bring no gain (a perfect network does not need any data redundancy). When the quality of the link goes down, using one FEC packet for 10 data packets requires transmitting less packets than when using no FEC. Doubling the number of packets (transmitting 10 FEC packets for 10 data packets) will result in good performances on very low quality links (loss ratio between 30% and 50%) and will almost never require more than 25 packets, while pure retransmission can use more than 35 packets. These figures are comparable with the results obtained by implementing [17].

3 Adding fault tolerance into GLADE

3.1 Single point of failure

Up to now, GLADE had one single point of failure called the *boot server*. This server is located on one partition, called the *main partition*, because it holds the main subprogram of the distributed application when one exists[3].

The boot server fulfills two different functions. The first one consists in assigning an unique partition ID (see annex E of [8] for a definition of partition ID) to a partition joining a distributed program. Figure 6 shows a sample of the dialog between the newly created partition and the boot server, whose address is either present in the application configuration file or provided on the partition command line.

The second function of the boot server, as shown on figure 7, is to register and later publish the physical location of partitions, as well as the list of `Remote_Call_Interface` package bodies that have been elaborated on each partition.

Unfortunately, this boot server has been a major weak point in GLADE while trying to build fault tolerant programs because as soon as this partition dies, no new partition can connect to the distributed application (for example, it was impossible to restart a died partition after the disappearance of the boot server). For this reason, we decided

[3]In Ada 95, a program can run as a result of elaborating its partitions, the main subprogram being optional (see [8] section 10.2(34)).

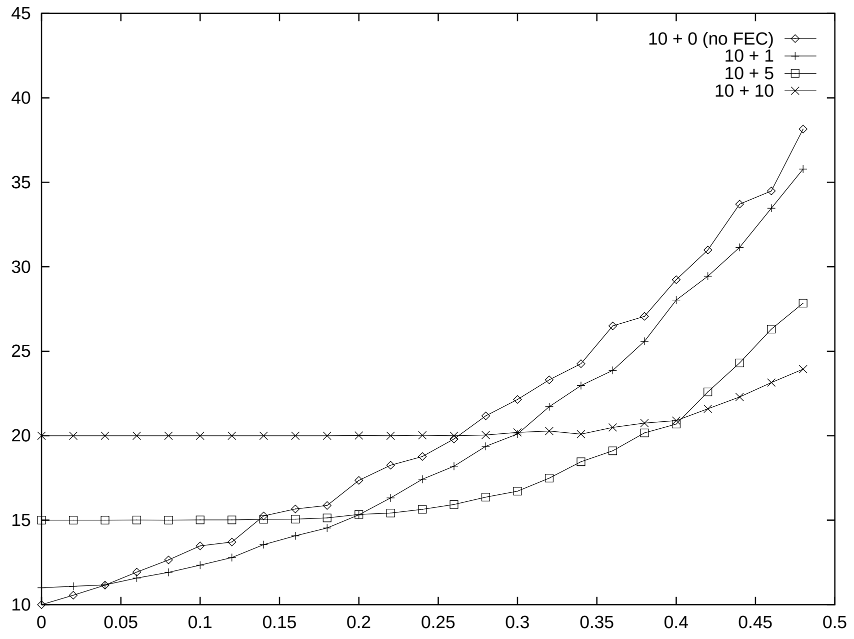

Figure 5: Number of transmitted packets according to the loss probability

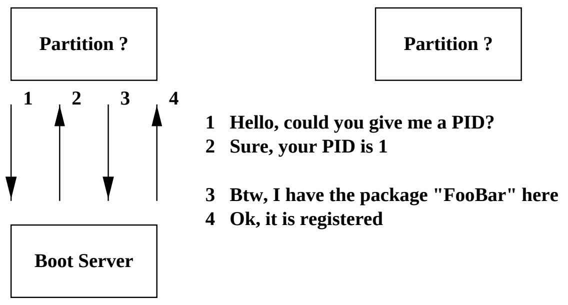

Figure 6: Partition ID and name servers

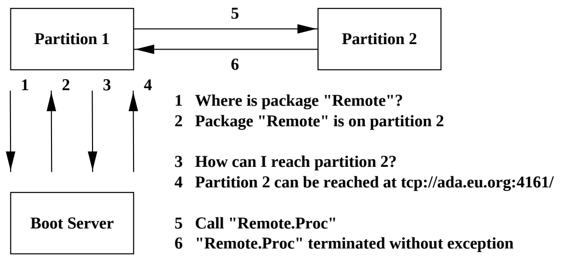
Partition 1
5
6
Partition 2
1
2
3
4
1 Where is package "Remote"?
2 Package "Remote" is on partition 2
3 How can I reach partition 2?
4 Partition 2 can be reached at tcp://ada.eu.org:4161/
Boot Server
5 Call "Remote.Proc"
6 "Remote.Proc" terminated without exception

Figure 7: Package to partition mapping

to use IP multicast with forward error correcting along with distributed semaphores to share the crucial data between several partitions, in order to survive the death of the boot server.

3.2 Sharing the data

Not only must several partitions share the data, but they also have to agree on which one of them will assign a partition ID when a new partition connects to the distributed application. This partition ID must be unique through the whole distributed application, so it is not possible to have any random partition just pick an unused partition ID and propagate it to the others without using some kind of election or strict locking.

For this purpose, we chose to use an algorithm based on the one described by Kai Li and Paul Hudak for implementing distributed semaphores ([11]). The name server problem is then trivially solved, because the request for a partition ID will be immediately followed by the registration of the `Remote_Call_Interface` packages that have been elaborated on the newly created partition.

3.3 The Li-Hudak algorithm

The algorithm introduces the notion of *probable owner* of a semaphore. At the beginning, each node points onto the creator of the semaphore.

Each time a node wants to acquire the semaphore when it does not own it, it sends a request to the probable owner (as it knows it) and records the probable owner as itself (since it will eventually get the semaphore).

Each time a node P receives a request for the semaphore from node N, it will act accordingly to table 1. The main idea behind this algorithm is that each time node P receives a request from node N, it will consider that node N is the

Probable owner (for node P)	**P uses semaphore**	**Action**
Node P	No	Give semaphore and set probable owner to N
Node P	Yes	Promise semaphore to N and set probable owner to N
Node X	No	Forward request to X and set probable owner to N
Node X	Yes	Forward request to X and set probable owner to N

Table 1: Request forwarding algorithm when node N requests the semaphore from node P

new probable owner of the semaphore, regardless of whether it was successful in giving the semaphore or not.

We trivially see that if we have a set of n nodes and reliable FIFO communication protocols (such as TCP), each request will be forwarded at most $n-1$ times before reaching the real owner of the semaphore and that no loops can be created. We can see an example of this algorithm on figures 8 and 9: on figure 8, the node N_1 is requesting the semaphore which is currently owned by node N_5. Node N_1 will send a request to node N_2 and set the probable owner to itself. Node N_2 will forward the request to node N_3 and set the probable owner of the semaphore to be N_1, and so on.

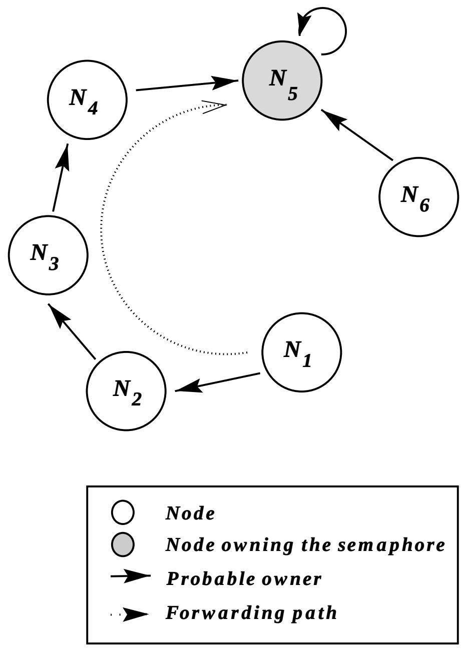

Figure 8: Node N_1 is requesting the semaphore

Whenever node N_1 obtains the semaphore, all the nodes but node N_6 (which was not on the forwarding path) correctly have N_1 as the probable owner of the semaphore. It means that if any of them needs the semaphore, only one request will be necessary (see figure 9).

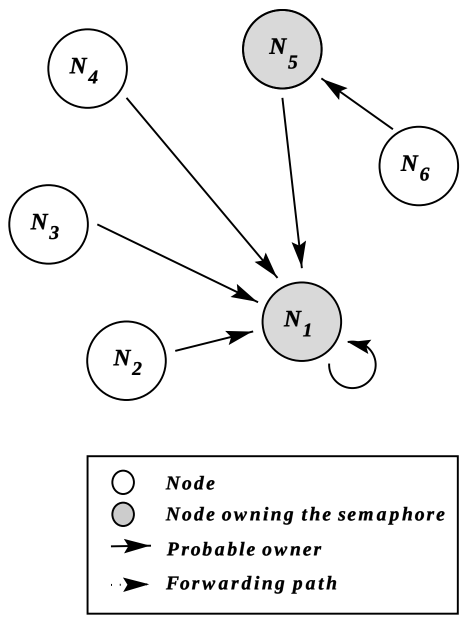

Figure 9: Node N_1 got the semaphore

3.4 Ada implementation of the Li-Hudak algorithm

A prototype implementation has been designed and coded by ENST students for the Mururoa project in late 1997 and is described in [2]. It was a quite tricky task because at this time GLADE was lacking the possibility of declaring distributed objects in `Remote_Types` package. It was then impossible to have a pointer to a distributed object as a field of another distributed object, thus making it hard to implement chains of distributed objects.

As part of a constant effort to provide the Ada community with freely available Ada network components[4], the Network and Computer Science department of ENST took advantage of the new possibility of declaring chains of distributed objects to develop a set of packages offering distributed semaphores. The internal structure of a semaphore is straightforward and described in figure 10 (declaration of package `GLADE.Semaphores`). As we can see, the Ada model for declaring distributed pointers is very well suited for this kind of use.

A graphical tool has also been developed to follow the various steps performed by this algorithm both for debugging and teaching purposes. A snapshot of this tool is provided on figure 11. This program will be made publicly available some time in the future, as we got many requests for such a tool, which really eases the development of safe distributed applications. This affords us with a new debugging tool in addition to [21].

Another package called `GLADE.Valued_Semaphores` implements a distributed semaphore that can also hold a value of any constrained data type, which can be changed only by the owner of the semaphore and is propagated along with it.

[4]The "Ada Network Components" page is located at `http://www.inf.enst.fr/ANC/`.

```
package GLADE.Semaphores is

   pragma Remote_Types;

   type Semaphore_Type is tagged limited private;

   procedure Initialize
     (Semaphore : access Semaphore_Type;
      Name      : in String);

   procedure P (Semaphore : access Semaphore_Type);

   procedure V (Semaphore : access Semaphore_Type);

[...]

private

[...]

   type String_Access is access String;

   type Status_Type is (Locked, Waiting, Unlocked);

   type Semaphore_Type is tagged limited record
      Status          : Status_Type;
      Probable_Owner  : Semaphore_Access;
      Promised_To     : Semaphore_Access;
      Barrier         : Positive;
      Name            : String_Access;
      Index           : Positive;
   end record;

end GLADE.Semaphores;
```

Figure 10: `GLADE.Semaphores` package declaration

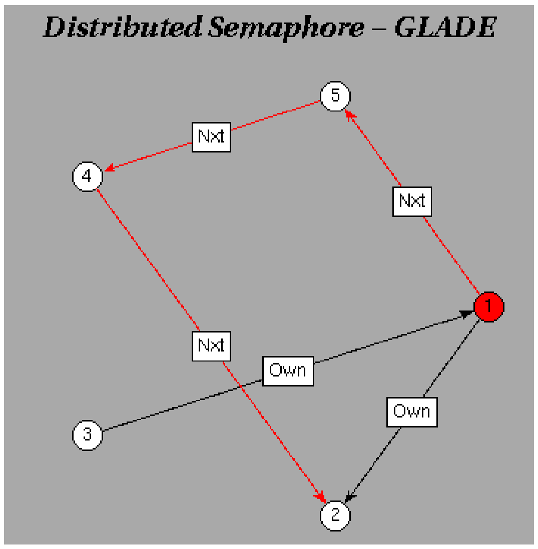

Figure 11: Graphical representation of Li-Hudak

3.5 Combining reliable multicast and distributed semaphores

We are currently using the following scheme to propagate the state of the partition ID and name servers:

- Each time some node wants to make a modification to the shared table, it first acquires the semaphore and gets the latest version of the table from the valued semaphore.
- It makes the modifications to the semaphore data, then broadcasts those new tables using IP multicast. We call this *early data transmission*, because the new data could be found anyhow in the semaphore itself.
- When some node needs to access data stored in the tables, it first checks whether its own copy has the necessary information. If it does not, it requests the semaphore to make sure that it has the latest version. If the requested data still cannot be found in the semaphore, it uses an extension to the distributed semaphore model which will cause the semaphore to be sent back whenever it has been taken and released. The requested information will thus be available to this node as soon as it is available in the semaphore. This part can of course be repeated several times until the data becomes available.

Except in the case where the information is not present in the table because it is not known yet, we have had very few cases of the local data being out-of-date in regard to the semaphore content, thus proving the efficiency of the FEC model.

We currently use an algorithm described in [13, 16, 3] to regenerate the semaphore when the partition holding it has crashed, but we are designing our own, which is more efficient at getting the latest available version of the tables.

4 Conclusion and work in progress

As of writing this paper, we have a working implementation of our model that will serve as a starting point for the next step. What we will do first is propagate only table updates in order to transmit large amounts of data with only a limited bandwidth. The algorithm used in xdelta ([12]) sounds like a good candidate. This will of course introduce ordering problems that are currently dealt of with a simple version number.

This is only a first step towards fault tolerance. We will then handle the case of duplicated remote procedure calls, to be able to deal with warm restart of partitions. Then we may consider the case of `Shared_Passive` packages, that could be implemented on top of such a shared memory area.

Since this is work in progress, the reader may expect to find papers on these topics in the near future.

Thanks

The authors would like to gratefully thank Thomas Quinot for his unvaluable comments on this paper, as well as Nadine Richard and Marie-Laure Tabary for their thorough review.

References

[1] S. Armstrong et al. Multicast transport protocol. Technical report, Internet Engineering Tasking Force, February 1992. RFC 1301, `http://www.ietf.org/`.

[2] Romain Berrendonner, Laurent Bousquet, Thomas Quinot, and Stéphane Thellier. Mururoa: An example of distributed mutual exclusion using distributed objects in Ada95. Technical report, École Nationale Supérieure des Télécommunications, December 1997.

[3] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, second edition, 1994.

[4] Bertrand Dupouy, Olivier Hainque, Laurent Pautet, and Samuel Tardieu. The SPIF project. In *Proceedings of AdaEurope'97*, London, UK, June 1997. Springer Verlag.

[5] Sidnie Feit. *TCP/IP: architecture, protocols, and implementation*. McGraw-Hill, 1993.

[6] Alexandre Fenyö, Frédéric Le Guern, and Samuel Tardieu. *Connecter son réseau d'entreprise à l'Internet*. Eyrolles, March 1997.

[7] Anthony Gargaro, Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. PARIS: Partitionned Ada for Remotely Invoked Services. In *Proceedings of AdaEurope'95*, Frankfurt, Germany, March 1995.

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

[9] Yvon Kermarrec, Laurent Nana, and Laurent Pautet. GNATDIST: a configuration language for distributed Ada 95 applications. In *Proceedings of Tri-Ada'96*, Philadelphia, Pennsylvania, USA, 1996.

[10] Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. GARLIC: Generic Ada Reusable Library for Interpartition Communication. In *Proceedings Tri-Ada'95*, Anaheim, California, USA, 1995. ACM.

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

[12] Josh Mac Donald. *xdelta, a very efficient delta generator*. `http://www.XCF.Berkeley.EDU/~jmacd/xdelta.html`.

[13] Sape Mullender. *Distributed Systems*. Addison-Wesley, second edition, 1993.

[14] OAR. *RTEMS User's Guide*. Online Applications Research Corporation, 1998. `http://www.oarcorp.com/`.

[15] Laurent Pautet and Thomas Wolf. Transparent filtering of streams in GLADE. In *Proceedings of Tri-Ada'97*, Saint-Louis, Missouri, USA, 1997.

[16] Michel Raynal. *Gestion des données réparties : problèmes et protocoles*. Collection de la Direction des Études et Recherches d'Électricité de France. Eyrolles, 1992.

[17] J. Rosenberg and H. Schulzrinne. An A/V Profile Extension for Generic Forward Error Correction in RTP. Technical report, Bell Laboratories, 1997.

[18] Edmond Schonberg and Bernard Banner. The GNAT project: A GNU-Ada 9X compiler. In *Proceedings of Tri-Ada'94*, Baltimore, Maryland, USA, 1994.

[19] Samuel Tardieu and Laurent Pautet. Au cœur de l'annexe des systèmes répartis. *La lettre Ada*. To be published.

[20] Samuel Tardieu and Laurent Pautet. Inside the Distributed Systems Annex. In *Proceedings of AdaEurope'98*, Uppsala, Sweden, 1998. Springer Verlag.

[21] Samuel Tardieu, Laurent Pautet, and Daniel Neri. Debugging distributed applications with replay capabilities. In *Proceedings of Tri-Ada'97*, Saint-Louis, Missouri, USA, 1997.