An Ada library to program fault-tolerant distributed applications

Conference Paper · June 1997

DOI: 10.1007/3-540-63114-3_21 · Source: dx.doi.org

4 authors, including:

Francisco Javier Miranda González
Universidad de Las Palmas de Gran Canaria
33 PUBLICATIONS **112** CITATIONS
SEE PROFILE

Alejandro Alvarez
Universidad Politécnica de Madrid
8 PUBLICATIONS **30** CITATIONS
SEE PROFILE

Sergio Arévalo
Universidad Politécnica de Madrid
78 PUBLICATIONS **806** CITATIONS
SEE PROFILE

An Ada Library to Program Fault-tolerant Distributed Applications*

F. Guerra, J. Miranda, A. Alvarez and S. Arévalo

University of Las Palmas de Gran Canaria and Technical University of Madrid
fguerra@cic.teleco.ulpgc.es, jmiranda@cma.ulpgc.es,
aalvarez@dit.upm.es, sarevalo@fi.upm.es

Abstract. This paper describes a library written in Ada which facilitates the construction of fault-tolerant distributed applications based on the active replication paradigm [18]. The library, called Group_IO [10], offers a simple interface to the implementation of reliable, atomic, causal, and uniform multicast. The work on Group_IO has been motivated by our experience with Isis [3] and similar reliable multicast frameworks. The library allows also client–server interactions where the client may be a group—this interaction is not supported by ISIS— and relies on an own consensus protocol [8, 9] to implement the uniform broadcast protocols. Group_IO is the base on which the programming language Drago [2, 15, 16] has been implemented, however it does not require Drago for its use.

Keywords: Distributed Systems, Fault-Tolerant Systems, Ada, Isis.

1 Introduction

The increasing dependence of modern society on computer systems calls for increasing degrees of reliability which become very expensive to implement with traditional hardware and software techniques. In particular, the use of ad-hoc replicated hardware to mask out failures requires special components with costs much higher than the ones of standard, mass produced hardware. As a result, the use of modern solutions in which the tolerance to hardware failures is obtained by means of specialized software running on top of standard, inexpensive hardware, is attracting a considerable degree of attention. However, the construction of this specialized software is a rather complex task, and so the need for software libraries that support these new programming paradigms arises.

The basic approach to fault-tolerance using standard hardware components is the use of distributed systems with hardware and software replication. The two main software techniques used there are the primary-backup approach, and the active replication paradigm [19]. Compared with the primary-backup approach, the active replication technique offers the additional advantage that it

* *This work has been partially funded by the Spanish Research Council (CICYT), contract numbers TIC94–0162–C02–01 and TIC96–0614.*

allows for continuous service in the presence of failures. That is, the system can continue giving service without the need to interrupt for any length of time to be reconfigured or in any way recover it after a failure.

In order to build programs with active replication and minimal additional effort from the programmer, there is a need for transparent mechanisms to handle the communication when a group of replicas receives a service request or requests an external service [13]. In particular, for every single message sent to a group of process replicas, the underlying system should transparently ensure that the message is replicated and a copy of it sent to each replica of the process—this is known as "1-to-n" communication. When all interaction among processes takes the form of message exchanges, all replicas of the same process must receive the same messages, even in the presence of (partial) failures—this is known as "all-or-none" communication—and in the same order. Symmetrically, (replicated) messages sent by the replicas themselves shall be filtered so that only a single copy of each replicated message is actually issued to the rest of the system—this is known as "n-to-1" communication.

The problem of "1-to-n" communication has been discussed at length in numerous publications, where is has received the name of reliable broadcast [4] [17] [14]. By contrast, references to "n-to-1" communication cannot be easily found.

It has been proven that uniform reliable—atomic—and totally ordered broadcast is equivalent to distributed consensus [11]. In the consensus protocol, a number of processes start each one proposing a possibly different value, and at the end of the protocol all (correct) processes end up agreeing on the same value, even if some of them happen to fail during the execution of the protocol itself. To see how both mechanisms are equivalent, one only needs to consider a (sequence of) consensus where the values to agree upon are the actual messages the different processes wish to broadcast, and to understand the agreement to select a particular message as the delivery[2] of that message in all the processes involved in that consensus. In addition, when one process finds that its message has not been selected in a consensus, it just stubbornly insists on proposing the same message until it eventually gets selected. As a result, all processes involved in the (sequence of) consensus end up receiving the same messages and in the same (total) order.

The work on Group_IO has been motivated by our experience with Isis, with similar reliable multicast frameworks, and with the development of different consensus protocols. By contrast, systems such as PVM [6] and MPI [7] had no influence on Group_IO as they only provide a basic broadcast service, without features like causality, order or atomicity, which are needed to program replicas in the fault-tolerant active replication model.

After this brief introduction, in the next two sections we present a short description of ISIS and Drago, respectively. We then have a section on the interface

[2] For other than the basic broadcast, *delivery* of messages is an event different from *reception*; the distinction is needed in order to enforce the required message order, in spite of the actual transmission times.

offered by Group_IO, followed by another section with some programming examples. Three more sections then discuss implementation aspects of Group_IO, and its relation with Ada 95 and Drago. The paper closes with some conclusions, and with references to related work.

2 ISIS

ISIS [3] is a toolkit that goes a long way in the active replication line just described and which has been the original inspiration for Group_IO. In ISIS programmers can define groups of processes and then refer to them by a single name. Communication with a group of processes is by means of (different versions of) *reliable*[3] *broadcast*[4], which can be used to implement replicated (as well as *cooperative*[5]) process groups.

However, from our experience with ISIS the system has three major drawbacks. First, ISIS broadcast is not *uniform*, that is, there is no guarantee that non-failed processes receive a message which has anyhow been received by a process failing subsequently to the reception of that message. And the problem with this approach is that if the failed process has taken any actions after receiving that message and before failing, its remaining replicas will be out of sync with it. As a consequence, it is close to impossible to implement active process replication in ISIS along the lines described above.

The second problem with ISIS is that it does not support full n-to-1 communication[6]. Last but not least, ISIS provides no linguistic support. In fact, ISIS is just a collection of libraries written in C, and as such its use leads to code which is both complex and error-prone.

3 Drago

Drago[16] is an experimental language developed as an extension of Ada for the construction of fault-tolerant distributed applications. The hardware assumptions are a distributed system with no memory shared among the different nodes, a reliable communication network with no partitions, and *fail-silent* nodes. (That is nodes, which once failed are never heard from again by the rest of the system.)

The language is the result of an effort to impose discipline and give linguistic support to the main concepts of ISIS[3], as well as to experiment with the group

[3] What ISIS calls *reliable* is actually called *atomic* by other authors to reflect its "all-or-none" property.

[4] Actually, a kind of *multicast* remote procedure call, but we will use here the term *broadcast* to follow ISIS convention.

[5] Member processes of a cooperative group usually do not perform exactly the same function, and make use of this fact to "cooperate" in the provision of one or more services.

[6] In particular, when a replicated process group issues a call to another process, be it a single process or a group, as many calls as group members are issued.

communication paradigm. To help build fault-tolerant distributed applications, Drago explicitly supports two process group paradigms, *replicated process groups* and *cooperative process groups*. Replicated process groups allow for the programming of fault-tolerance applications according to the active replication model[18], while cooperative process groups permit programmers to express parallelism and so increase throughput.

A process group in Drago is actually a collection of *agents*, which is the way processes are called in the language. Agents are rather similar in appearance to Ada tasks (they have an internal state not directly accessible from outside the agent, an independent flow of control, and special operations named *entries*) although they are the unit of distribution in Drago and so perform a role similar to Ada 95 active partitions and Ada 83 programs. Each agent resides in a single node of the network, although several agents may reside in the same node. A Drago program is composed of a number of agents residing at a number of nodes.

Aside from distribution, the main difference they have with Ada tasks is that calls to its entries are automatically ordered by the underlying Drago global runtime message system to enforce reliable, causal, uniform coordination among the agents of the same group. This is actually the essence of Drago, and what makes it most useful.

4 Group_IO Interface

Group_IO is a library built as a generic Ada package that provides operations and types to perform distributed client-server interactions among Ada programs organized as groups according to the active replication model. In this model clients—either a single Ada program or a group of them—issue requests to servers made up of groups of Ada program replicas—running in different network nodes—and then wait for replies. Group_IO transparently masks out possible failures of nodes running the Ada program replicas.

Group_IO provides a generic interface that expects the user to define the maximum size of requests and replies, the retransmition time, and other parameters that depend on the particular system and distributed application at hand. Basic types provided by a generic instance of Group_IO are:

```
subtype T_Data is STRING (1 .. Max_Length_Data);
subtype T_Name is STRING (1 .. Max_Length_Name);
type T_Group_Id is private;
type T_Request_Id is private;
```

All the information contained in the requests and replies sent through the network are strings of type *T_Data*. It is the responsibility of user programs to know how to use the messages delivered and to perform type conversion when needed. Group names are strings of type *T_Name*. Group_IO also provides two types to declare handlers for groups and requests, respectively: *T_Group_Id* and *T_Request_Id*.

The way to use Group_IO depends on whether the user software behaves as a client, a server, or a replica.

Client interface:

- Before a client requests a service to a server group, it must start with a call to join that group as a client:

```
procedure Join_Group_Client (Grp_Name : in  T_Name;
                             Grp      : out T_Group_Id);
```

  Join_Group_Client creates the data structures and tasks associated with the client role and returns a group handler (*Grp.*) The exception *Inactive_Group* is raised in case there is no group named Grp_Name.
- After a user program has obtained a group handler *Grp*, it can send a request to the associated group:

```
procedure Send_Request  (Grp  : in  T_Group_Id;
                         Mess : in  T_Data;
                         Req  : out T_Request_Id);
```

  Send_Request blocks the caller only until the request arrives to all live members of server group *Grp*, and then it returns the request handler *Req*. The exception *Inactive_Group* is raised if the caller is not a client of group *Grp* or when no members of the group *Grp* are alive anymore.
- The number of replies is not fixed because members of a group may fail. A user program can get the number of pending replies—each group member gives its own reply—with the function:

```
function Replies_Number (Req : T_Request_Id) return Natural;
```

  The exception *Invalid_Handler* is raised when *Req* is an invalid request handler.
- After *Send_Request* returns the request handler, the user program can retrieve the replies received—all members of the server group reply—one by one:

```
procedure Receive_Reply(Req  : in out T_Request_Id;
                        Mess : out    T_Data);
```

  The exception *Invalid_Handler* is raised when Req is an invalid request handler; the exception *No_Replies* is raised when all replies from live members have already been delivered.
- A user program only gets the number of replies it wishes. In particular, it can use a reply before getting the next one, and this will be the usual case in which the first reply to arrive will be used and all the rest will be discarded. User programs can indicate that they do not wish to receive any more of the replies belonging to a certain request calling the procedure:

```
procedure Close_Request(Req : in out T_Request_Id);
```

  Close_Request either marks the handler *Req* as invalid or raises the exception *Invalid_Handler* if it is already invalid before the call.

Server interface:

- All server members must first join the group:

```
procedure Join_Group_Server(Grp_Name : in   T_Name)
                           Grp      : out  T_Group_Id);
```

  Join_Group_Server creates the data structures and tasks associated with the server role and returns a group handler. The exception *Inactive_Group* is raised if there is no group named *Grp_Name*.
- Every member of the server group can get the next request made to the group:

```
procedure Receive_Request(Grp  : in  T_Group_Id;
                          Req  : out T_Request_Id;
                          Mess : out T_Data);
```

  Receive_Request returns the handler *Req* associated with the request *Mess*. This handler is used later to send the associated reply. The exception *Inactive_Group* is raised when the user code calling *Receive_Request* is not a member of server group *Grp*.
- Servers send its replies associated to a request with the procedure:

```
procedure Send_Reply(Req  : in out T_Request_Id;
                     Mess : in     T_Data);
```

  After the reply is sent, the handler becomes invalid—every group member can only send a single reply. The exception *Invalid_Handler* is raised when the handler *Req* is invalid.

Replicated client interface: To implement a fault-tolerant service by means of a group of replicas we should only use the server interface. However, when this group of replicas needs to request a service—the group of replicas can be client of any other group—it is necessary to add to the client interface some operations where the client's group of replicas is referenced.

- Every replica must still call first *Join_Group_Server* as before. However, before the group of replicas issues a request, every replica must execute the next procedure, passing the replica group handler *Replica_Grp* and the name of the server group *Grp_Name* as parameters:

```
procedure Join_Group_Client(Replica_Grp : in  T_Group_Id;
                            Grp_Name    : in  T_Name;
                            Grp         : out T_Group_Id);
```

  Join_Group_Client creates the data structures and tasks associated with the replicated client role and returns a group handler. The exception *Inactive_Group* is raised when the caller is not a member of the group *Replica_Grp* or when there is no group named *Grp_Name*.

– When a replica sends a request to a server group, it must also pass the handler associated with the group of replicas.

```
procedure Send_Request(Replica_Grp : in  T_Group_Id;
                       Server_Grp  : in  T_Group_Id;
                       Mess        : in  T_Data;
                       Req         : out T_Request_Id);
```

Again, *Send_Request* blocks the caller only until the request arrives to all live members of server group *Grp*, and then returns the request handler *Req*. The exception *Inactive_Group* is raised when the caller is not a member of group *Replica_Grp*, the replica group is not a client of group *Grp*, or all members of group *Grp* have already failed.

– It is crucial that all the replicas have the same code, perform the same actions, and go through the same sequence of states, and in the same order. Group_IO delivers the same sequence of requests—server role—and replies—client role—to all replicas. However, it is necessary to call the next procedure to find out the kind of the next message delivered because requests and replies are delivered by different procedures[7].

```
type T_Operation_Id is  (Replica_Request, Replica_Reply, Final_Reply);

procedure Next_Operation(Replica_Grp : in  T_Group_Id;
                         Req         : out T_Request_Id;
                         Operation   : out T_Operation_Id);
```

Next_Operation returns the handler *Req* associated to the next request (or reply) and the kind of operation—*Operation* is equal to *Replica_Request* or *Replica_Reply*. This handler is also returned when there are no more replies associated with a request—*Operation* is equal to *Final_Reply*, and so does not need to call *Replies_Number* repeatedly. The exception *Inactive_Group* is raised when the caller is not a member of the group *Replica_Grp*.

5 Programming with Group_IO

Every user program must create its own instance of Group_IO, passing generic actual parameters defining its system and distributed application. For example:

```
with Group_IO;
package My_Group_IO is
        new Group_IO(Max_Length_Mess       => 128,
                     Max_Groups            => 5,
                     Max_Members_Per_Group => 15,
                     Max_Asinc             => 4);
```

My_Group_IO is an instance that defines the maximum length of a request or reply message; the maximum number of groups; the maximum number of

[7] This is particularly important for those cases in which the programming language used includes non-deterministic constructs, as is the case with Ada tasks; all non-determinism must then be resolved in the same way for all process replicas.

members per group; and the maximum number of requests that can be pending to be delivered to any server group member. Additional parameters of Group_IO take the default formal parameter values. Let's see some examples of use of *My_Group_IO*:

- **Server**. A server has a loop where it gets the requests, performs the service, and sends the associated reply.

```
with My_Group_IO;
use My_Group_IO;

procedure Server is
  Grp_Id    : T_Group_Id;
  Req_Id    : T_Request_Id;
  Req_Mess,
  Ans_Mess  : T_Data;
begin
  Join_Group_Server("G1", Grp_Id);             -- It is member of group G1
  loop
    Receive_Request(Grp_Id, Req_Id, Req_Mess); -- The next request is delivered
    -- ...                                     -- THE SERVICE IS PERFORMED
    -- ...                                     -- THE REPLY IS PLACED IN Ans_Mess
    Send_Reply(Req_Id, Ans_Mess);              -- The reply is sent
  end loop;
end Server;
```

- **Client**. This example presents the code associated with a client that sends a request and waits for all replies.

```
with My_Group_IO;
use My_Group_IO;

procedure Client is
  Grp_Id    : T_Group_Id;
  Req_Id    : T_Request_Id;
  Req_Mess,
  Ans_Mess  : T_Data;
begin
  Join_Group_Client("G1", Grp_Id);             -- It is client of group G1
  -- ...                                       -- THE REQUEST IS PLACED IN Req_Mess
  Send_Request(Grp_Id, Req_Mess, Req_Id);      -- The request is sent
  While Replies_Number(Req_Id)>0 loop          -- While there are replies
        Receive_Reply(Req_Id, Ans_Mess);       --     The next reply is delivered
        -- ...                                 --     THE REPLY CAN BE USED
  end loop;                                    -- end
  Close_Request(Req_Id);                       -- The handler is marked as invalid
end Client;
```

- **Replicated client**. Let's now see the code of a group of replicas which is simultaneously server and client of other groups.

```
with My_Group_IO;
use My_Group_IO;

procedure Client_Server_Replica is
  Replica_Grp_Id, Server_Grp_Id        : T_Group_Id;
  Req_Id                               : T_Request_Id;
  Replica_Req_Mess, Req_Mess, Ans_Mess : T_Data;
  OP                                   : T_Operation_Id;
begin
```

```
Join_Group_Server("G1", Replica_Grp_Id);      -- It is member of replicas group G1
Join_Group_Client(Replica_Grp_Id,             -- The group G1 is a client of
                  "G2",                       -- group G2
                  Server_Grp_Id);
-- ...                                        -- THE REQUEST IS PLACED IN
                                              -- Replica_Req_Mess
Send_Request(Replica_Grp_Id, Server_Grp_Id,
             Replica_Req_Mess, Req_Id);       -- The request is sent
loop
   Next_Operation(Replica_Grp_Id, Req_Id, OP);-- The next event is received
   case OP is
     when Replica_Request =>
           Receive_Request(Replica_Grp_Id,    -- A Request is delivered
                           Req_Id,
                           Req_Mess);
           -- ...                             -- THE SERVICE IS PERFORMED
           -- ...                             -- THE REPLY IS PLACED IN Ans_Mess
           Send_Reply(Req_Id, Ans_Mess);      -- The reply is sent

      when Replica_Reply   =>
            Receive_Reply(Req_Id,             -- The next reply is delivered
                          Ans_Mess);
            -- ...                            -- THE REPLY CAN BE USED

      when Final_Reply     =>
            Close_Request (Req_Id);           -- The handler is invalid
    end case;
  end loop;
end Client_Server_Replica;
```

Next_Operation is also used when the group of replicas is just a client of various server groups because it must know the request handler associated with every delivered reply.

6 Group_IO Implementation

Group_IO is currently implemented over an Ethernet with Sun Sparc stations and uses Paradise [5] as its basic communication service—Paradise only blocks the task issuing an IO operation and not the whole Ada process. The protocols implemented in Group_IO assume a distributed hardware system with no memory shared among the different nodes, a reliable communication network with no partitions, and *fail-silent* hardware nodes; namely, when one of the nodes fails the rest of the system never hears from it again.

Each user program can be seen as a logical machine in the distributed system. Several logical machines may execute on the same physical machine, but each needs its own instance of Group_IO— see figure 1.

The Group_IO body is composed of three levels: Medium Access Level, Group Protocol Level, and User Operation Level. The Medium Access Level uses Paradise to get access to Berkeley sockets. The Group Protocol level uses the services of Medium Access Level to implement the multicast protocols [10] and provides the services required by the User Operation Level. Finally, the User Operation Level implements the user program interface.

Each Group_IO instance has common information used by all the three levels. For every group the user program belongs to, either as a client or as a server,

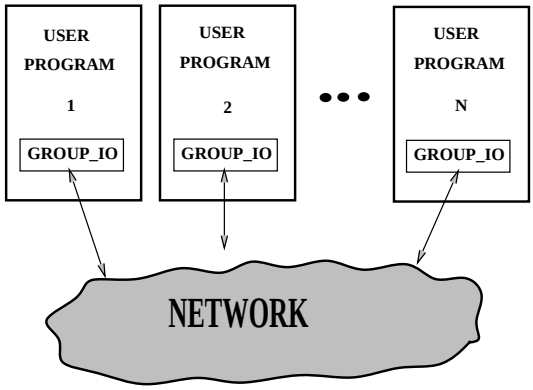

Fig. 1. User programs connected by Group_IO.

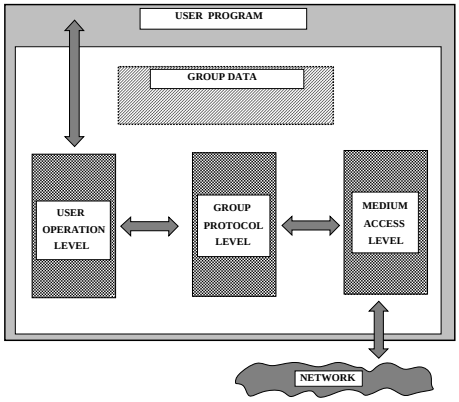

Fig. 2. Internal levels of Group_IO.

this information describes:

- Name of the group.
- Identity of the user program to communicate with the group.
- Role(s) of the program within the used group: client, server, or replica.
- View of the group (number of members and linear order of each one.)
- Pointers to the tasks that execute the different role(s) the user program has within the group.

There are two task types in the Group Protocol Level: the type T_Client executes the role associated with a client and the type T_Member defines the role associated with a server that is member of a group. As an example, the user program with the Group_IO instance shown in figure 3, is both a client of group A and a member of group B. Furthermore, this program communicates also with the rest of members of group B because t is also a client of this group—the

members of group B may cooperate to give the requested services, this is known as intragroup interaction.

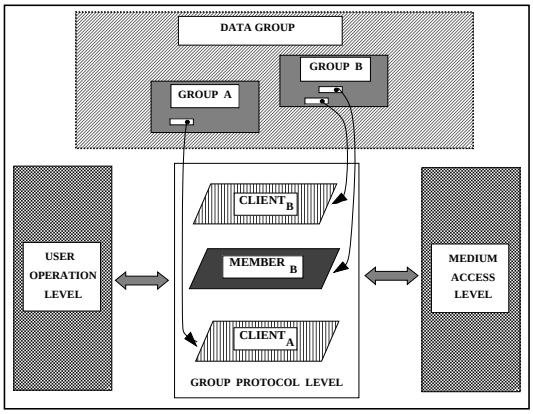

Fig. 3. Example of Group Protocol Level

Currently the groups configuration is static and it is specified by means of two files:

- *Logic_Names.Dat* defines the names associated with every user program, the Internet address (IP) of the machine where the program executes, and the port address (UDP) through which the program communicates. Each line of this file has the following format:

```
PROCESS_NAME, INTERNET_ADDR, PORT_NUMBER
```

- *Groups_Configuration.Dat* defines the names of the groups and the members of every group according to the next syntax:

```
GROUP_NAME := PROCESS_NAME {,PROCESS_NAME};
```

7 Group_IO and Ada 95

Group_IO has been implemented in Ada 83 and has been used with Ada 83 programs. We believe that some features of Ada 95 can be used to improve the implementation of Group_IO, protected objects in particular. What is not clear to us is whether we can take advantage of the distributed partition paradigm instead of a socket library. However, we don't see any problems for Ada 95 programs to make use of the Group_IO library.

A different issue is how distributed programs built with Ada and Group_IO (let's call them Group_IO programs) compare with distributed programs built with Ada 95 distributed active partitions.

One important difference between Group_IO programs and active partitions is that the first accept services explicitly issuing a *receive request* as part of their flow of control, while the second export *passive* subprograms declared in their RCI package specifications. This difference makes the first kind of programs deterministic in its behavior, while the behavior of the second ones is dependent on the runtime system. As we will see below, determinism is essential in the active replication model.

The main difference between Group_IO programs and the distributed programming model of Ada 95 is that Group_IO provides direct support for the active replication model in order to build fault-tolerant applications. Where Ada 95 provides a single remote call, Group_IO transparently gives programmers multiple *send_requests* to all the programs of a replicated group. And more important, the multiple *send_requests* are automatically coordinated so that all programs of the same group are guaranteed to receive the same *requests*, and in the same (causally consistent) order, even in the presence of hardware node failures midway in the sequence of calls. And because replicated Group_IO programs have a deterministic behavior all members of the same group go through the same sequence of internal states, and so give exactly the same replies to all incoming calls. This way live members of a group can mask out transparently the possible failure of other group members, something that cannot be obtained in Ada 95 without a considerable effort from the part of programmers.

On the question of what is the relation between Group_IO and the Distributed Systems Annex of Ada 95 and, in particular, on whether Group_IO can be integrated into the PCS to implement transparent replication of active partitions in Ada95, the answer is yes (with one natural caveat.)

There would be no need to change the specification of System.RPC, nor the compiler or the language itself, only rewrite the implementation of System.RPC so that it calls Group_IO when needed, that is, when the actual remote call is directed towards a procedure of a replicated partition. Whether a certain partition is replicated or not would be decided at configuration time (after compile time and before link time) and that information could be stored in a configuration file, from where the code of System.RPC would retrieve it at run-time.

It is clear nevertheless, that replicated partitions in no case could be serving more than one request at the same time. This is a restriction required not by Group_IO but by the replicated state-machine model, in orden to guarantee replica determinism.

8 Group_IO and Drago

The language Drago uses Group_IO as its communication subsystem and so supports the same group mechanisms as Group_IO. This similarity allows Ada and Drago programs to interoperate in a straightforward manner. For example, we can have one or several fault-tolerant services implemented in Drago executing in a distributed system, and then compile and run "Ada-with-Group_IO" clients that use those services in a manner analogous to the one described in section 5

above. The only thing the clients need to know are the logical names of the Drago groups that implement those services—and those names are within the configuration files. In this way the Drago code may be as complex as the application requires while the interface to the Ada clients can be kept quite simple.

9 Conclusions

This paper has described a reliable multicast library written in Ada, Group_IO, that can be used to easily build fault-tolerant distributed applications, themselves also written in Ada. The programming paradigm supported is the active replication state-machine model.

The current implementation of Group_IO runs on a SUN/OS network with the SUN-Ada compiler, and provides reliable atomic broadcast using an original consensus protocol[1][8][9]. All the communications are based on standard TCP/IP protocols and use the PARADISE[5] library of UNIX kernel calls.

The implementation is rather crude as far as efficiency goes—mainly due to the use of TCP/IP protocols—and no measures of performance have yet been taken. The configuration and load work is currently performed by hand with a minimal support from the file system, basically a configuration file kept at all participating nodes. We are reimplementing Group_IO on top of GNAT with Linux, and investigating how to handle dynamic groups where members enter and leave groups at run-time.

The interface proposed is the result of an effort to impose discipline and give an Ada binding to the main concepts of ISIS [3]. This interface also permits client-server interactions where the client may be a group—this interaction is not supported by ISIS—and it has been designed and implemented to support the code generation for the Drago language [2, 15, 16].

At any rate, the funcionality of Group_IO does not relate much to Ada 95 nor Ada 83, but to the future of Ada instead, i.e. Ada 0X. It is likely that by the time of the new revision of Ada, aspects such as fault-tolerance by replication and reliable broadcast will have to be considered into the new standard. And so the interest of the Ada community to start experimenting with these techniques.

10 Acknowledgments

We wish to thank the members of the Distributed Systems Seminar in the Technical University of Madrid for their help in clarifying the ideas contained in this paper.

References

1. Arévalo, S. and Gehani N. H. 1989. Replica Consensus in Fault Tolerant Concurrent C. Technical Report AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

2. Arévalo, S., Álvarez, A., Miranda, J. and Guerra, F.: A Fault-tolerant Programming Language Based on Distributed Consensus, *Cabernet'94 Workshop*, Dublin (March 1994)
3. Birman, K., R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The Isis System Manual. Version 2.1.* September 1990.
4. Chang, J. M. and Maxemchuck, N. 1984. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3), pages 251–273.
5. Courtel, N., *PARADISE: Package of Asynchronous Real-Time Ada Drivers for Interconnected Systems Exchange, version 3.2.* GNU (January 1993).
6. Geist, A. et al.: *PVM: Parallel Virtual Machine; A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Mass. (1994)
7. Gropp, W., Lusk, E., and Skjellum, A.: *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Mass. (1994)
8. Guerra, F., Arévalo, S., Álvarez, A., and Miranda, J. A Distributed Consensus Protocol with a Coordinator. *IFIP International Conference on Decentralized and Distributed Systems ICDDS'93.* Palma de Mallorca (Spain). September 1993.
9. Guerra, F., Arévalo, S., Álvarez, A., and Miranda, J. A Quick Distributed Consensus Protocol. *Microprocessing and Microprogramming 39* (1993) pp.111–114.
10. Guerra, F. 1995. *Efficient Consensus Protocols for Distributed Systems.* Doctoral Dissertation. Technical University of Madrid. (In Spanish.)
11. Hadzilacos V. and Toueg, S. 1993. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley.
12. Intermetrics, Inc. 1995. *Ada 95 Language Reference Manual.* Intermetrics, Inc., Cambridge, Mass. (January).
13. Liang, L., Chanson, S.T., and Neufeld, G.W.: Process Groups and Group Communications: Classification and Requirements. *IEEE Computer.* (February 1990)
14. Malki, D., Amir, Y., Dolev, D., and Kramer, S. 1994. The Transis approach to high available cluster communication. Technical Report CS-94-14, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.
15. Miranda, J. 1994. *Drago: A Language to Program Fault-tolerant and Cooperative Distributed Applications.* Doctoral Dissertation. Technical University of Madrid. (In Spanish.)
16. Miranda, J., Alvarez, A., Arévalo, S. and Guerra, F. Drago: An Ada Extension to Program Fault-Tolerant Distributed Applications. Proceedings of the *Reliable Software Technologies—Ada-Europe'96 Conference*, LNCS 1088, Springer Verlag.
17. Moser, L., Amir, Y., Melliar-Smith, P., and Agarwal, D. 1994. Extended Virtual Synchrony. In *IEEE 14th Intl. Distributed Computing Systems*, pages 56–67, June.
18. Schneider, F.B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, **22**(4), December 1990.
19. Guerraoui, R. and Schiper, A. Fault-Tolerance by Replication in Distributed Systems. Proceedings of the *Reliable Software Technologies—Ada-Europe'96 Conference*, LNCS 1088, Springer Verlag.

This article was processed using the LaTeX macro package with LLNCS style