Fault Tolerance in Distributed Ada 95

Thomas Wolf
Software Engineering Laboratory
Swiss Federal Institute of Technology
CH–1015 Lausanne, Switzerland
wolf@di.epfl.ch

Abstract

In this paper we present a project to provide fault tolerance in distributed Ada 95 applications by means of replication of partitions. Replication is intended to be largely transparent to an application. A group communication system is used for replica management. We examine some of the possibilities for implementing such a system and highlight some of the difficulties encountered in the context of the programming language Ada 95.

Keywords: *Distributed Systems, Fault Tolerance, Replication, Group Communication, Ada 95*

1 Introduction[1]

Although Ada 95 includes extensive support for the development of distributed applications, the tremendously important issue of fault tolerance is not addressed: the language does not include direct constructs for the development of fault–tolerant applications besides the ubiquitous exceptions. This despite the fact that one of the major targets of Ada is the development of real–time and embedded systems, where efficient mechanisms for achieving fault tolerance are mandatory.

Implementing fault tolerance in Ada 95 is left to the application developers. This increases the complexity of the software and thus raises the costs and error potential and is generally seen as an unsatisfactory situation. This paper describes a project striving to augment Ada 95 by transparent replication to achieve fault tolerance against hardware failures (i.e., node crashes) in an application–independent way, thus freeing the application from having to deal explicitly with these issues.

Transparency can only be achieved in close cooperation with the compiler. For the realization of our approach, we use the GNAT compiler [12] together with the corresponding implementation of the PCS (see section 2), called Garlic [7], and a group communication system developed at EPFL (PHOENIX, [9]).

The rest of this paper is organized as follows: after briefly reviewing the model of distributed applications in Ada 95 in section 2, we introduce replication in sections 3 and 4. Sections 5 and 6 elaborate on some design possibilities for implementing replication in Ada 95. Sections 7 and 8 finally deal with work in progress and as yet unsolved problems.

2 Distributed Applications in Ada 95

A distributed Ada 95 application is a set of partitions distributed on various processing nodes in a network. Each partition is an implementation of the virtual node concept [11]. Partitions themselves can hold several library units, some of which constitute the interface towards other partitions. The process of mapping the partitions of a program to the actual physical nodes in a distributed system is called configuration and is done after compilation. This makes it possible to create different distributed configurations of the same program without source changes.

Partitions communicate among each other using remote procedure calls (RPC)[2]. Remote procedure calls are highly transparent to the application: only the interface packages of a partition have to be marked as such by pragmas. Pragma `Remote_Call_Interface` marks packages declaring subprograms that may be called remotely; these packages are called RCI packages. In order to send the parameters of an RPC over a network, they have to be transformed into a flat stream of bytes and converted back to the original structure on the receiving side. This marshalling and unmarshalling is performed by code inserted by the compiler on both the sender and the receiver partition for every subprogram that can be accessed remotely. These pieces of code are called calling stub and receiving stub, respectively. The lowly communication protocol handling needed to synthesize the high–level abstraction of RPC is completely hidden from the application layer in the Partition Communication Subsystem (PCS), part of the Ada Runtime Support.

1. Position paper submitted to the 8th International Real–Time Ada Workshop (IRTAW–8) , to be held in Ravenscar UK, April 8 – 11, 1997

3 Fault Tolerance by Replication

For our investigation of fault tolerance by replicating partitions we will only consider crash failures [6] of nodes. Also, we will at first only treat homogeneous distributed systems. Concerning the network, our assumptions are given by those made by the underlying group communication system we use for replica management, i.e., we assume an asynchronous network [4] with unreliable failure detectors [3].

In addition, the replication scheme should be as transparent as possible for the application since we consider replication a configuration concern. This explains our choice of replication as the basic technique for achieving fault tolerance instead of more application–specific techniques like recovery blocks [8] or atomic actions [13].

Given the above model of distributed applications, it seems clear to choose the partition as the unit of replication: as it already is the unit of distribution, this is the most natural choice. We view all replicas of a partition as members of a group and use a view synchronous group communication system like PHOENIX [9] to manage them. Using such a system greatly simplifies the task of synchronizing the replicas: it provides powerful communication primitives such as a totally ordered reliable multicast[1] and automatic view changes, i.e., group membership information. In this context, we consider a distributed fault–tolerant application a collection of groups instead of single partitions.

Replication of partitions introduces redundancy in the application to achieve fault tolerance. It is therefore no surprise that the main problem of replication is guaranteeing the consistency of the replicas. Generally, there are two different possibilities to organize the groups to maintain state consistency on all replicas.

In *active* replication, requests are executed on all replicas in the same order, which guarantees a consistent evolution of the state if the partition is a state machine [11]. Active replication can completely mask failures of group members, and it can even handle byzantine failures of a minority of the replicas.

The second approach is *passive* replication, where only one of the replicas (the primary) handles a request. Before returning the result to the client, it communicates any state change to the other group members (its backups), which then bring their state up to date. If the primary fails before completing the remote subprogram call, one of the backups newly assumes the role of the primary and re-executes the request.

1. Note that the solution to the consensus problem underlying a totally ordered multicast also solves a number of agreement problems we'd otherwise have to solve at a higher layer.

4 Architectural Overview

Our goal is to provide fault tolerance in distributed Ada 95 programs by transparent replication of partitions. Replication is defined only in the configuration phase, prior development phases should be largely unaffected. It should be possible to replicate partitions selectively, including specification of the characterizing parameters of replication of each partition: degree of replication, physical nodes to run the replicas on, whether active replication or a primary–backup scheme is to be used, etc. Of course this requires elaborate support in a flexible configuration language.

The implementation is based on a view of a distributed Ada 95 application as a collection of groups. All the replicas of a partition form a group, which to the rest of the application should be indistinguishable from a normal partition that is not replicated. Basic group management is done using a group communication system like PHOENIX, upon which a replica manager implements the higher level coordination of replicas. This replica manager is part of the PCS (Garlic, [7]) and therefore hidden from the application. As a consequence, the application layer has no knowledge of replication and does not have to be concerned with it — all the replica management is done inside the PCS.

5 Replica Organization

Replication of general Ada 95 partitions is not easy. The main difficulty encountered is that Ada partitions are inherently non–deterministic due to the behavior of task dispatching, queueing policies, and `select` statements. This poses some rather difficult problems in the context of replication since there is no way to guarantee that a re–execution of a request after a node failure will lead to the same sequence of state changes. For this reason we require (for the time being) that a replicated Ada partition be deterministic. Currently, this is achieved through the pragmas defined in the Real–Time Annex of the Ada 95 standard [5] that provide the means to force deterministic behavior, plus the requirement that a partition that is to be replicated use no timed entry calls[2].

Under the assumption that replicated partitions are deterministic, i.e., that they are state machines in the sense of [11], both active and primary–backup replication are possible. In active replication, the client broadcasts its RPC request to all replicas. The underlying group communication system ensures that all replicas receive all such requests in the same order (totally ordered multicast) and thus execute the requests in that same order. Since replicas are deterministic, they are guaranteed to perform the same sequence of state changes, and to produce the same result.

2. Since there is no global time in a distributed Ada 95 application, a timed entry call might or might not time out on different replicas.

The replica manager only must ensure that nested RPCs are executed only once and that the result of such a nested remote call is again broadcast to all group members. There is no need for auxiliary data structures for error recovery: if a node fails, it simply leaves the group, and the remaining replicas will complete the task. Using active replication, it is even possible to tolerate byzantine failures by only returning a result if at least a majority of the replicas in the group arrive at the same result. Replicas delivering some other result are considered faulty and henceforth excluded from the group.

With primary–backup replication, only one replica (the primary) executes a remote call. All state accesses as well as nested RPCs and their results are broadcast to the other replicas (the backups), where they are logged and the state changes duplicated. If the primary successfully completes handling of the remote call, the backups clear their logs for this call and the primary returns the result. If the primary fails while executing a remote call, one of the backups newly assumes the role of the primary. It begins re–executing the remote call, using its logged information to ignore any state access or nested RPC already made by the previous primary.

When a failed replica recovers, it will re–join the group during the elaboration of its PCS. At that moment, it will receive the current state from the remaining group members. Just when it is about to handle the first RPC request after its recovery, it will use the state information it received when joining the group to bring its own state up to date.

6 Identifying the State

Regardless of how the replicas are organized, there must be a way of recognizing the state of a partition and in particular state changes: in passive replication to inform the backup replicas, in active replication to transfer the current state when a replica first joins the group. In both cases, state transfers are needed when a failed replica of the partition re–joins the group.

Replicated Objects

Obviously we cannot simply view all global data as the state of a partition — the replica management system has no way of finding out neither when the state is changed nor even what that state is, i.e., which variables it comprises! Only through the use of controlled types with a suitable implementation for the `Initialize`, `Adjust`, and `Finalize` subprograms could we guarantee that the replica manager is informed of any state change. One could imagine a root type for replicated objects and require that all replicated objects have types derived from this root type:

```
with Ada.Finalization; use Ada.Finalization;
type Replicated is new Controlled with private;
```

The state of a partition would then be the set of replicated objects it contains.

Yet this approach is unsatisfactory for several reasons: first, it imposes a certain discipline on the application programmer and restricts the generality of the approach; second, the `Adjust` procedure is not called when only a component of a controlled object is changed; and finally, these three primitive operations would have to be re–implemented for each and every type derived from `Replicated` because the replica manager not only has to know that a change occurred, but also what the change was.

Replicated Packages

Another possibility is to require that the state of a replicated partition must be encapsulated in packages marked by a special pragma `Replicated`. While also not completely transparent to the application, such an approach fits well into the model for programming distributed applications defined in annex E of the language standard [5], where interface packages of a partition must be marked by certain pragmas.

A `Replicated` package in a non–replicated partition behaves exactly like a normal package. But if the partition is replicated in the configuration phase, the compiler can easily recognize such packages and generate appropriate code to record any state access. (This transformation is very similar to the way GNAT treats RCI packages to generate the stubs.) For this to be possible, some restrictions must be placed on `Replicated` packages. These restrictions shall guarantee that the state be properly encapsulated within the `Replicated` packages and that accesses are only possible through subprograms or the entries of protected objects declared in the package specifications. Furthermore, we require that `Replicated` packages themselves be deterministic.

To maintain this tight encapsulation, neither the specification nor the body of a `Replicated` package may depend upon a normal package: both must only depend upon `Pure`, `Remote_Types` or other `Replicated` packages.

Types used in the interface of a `Replicated` package must all be declared in `Remote_Types` packages. This is necessary because each access to the encapsulated state — each call to a subroutine or an entry of a protected object declared in the specification including the arguments of the call — must be sent to all backups. To do so, we must be able to marshal the arguments, and we can only guarantee this to work properly if these arguments are remote types.

These restrictions allow a `Replicated` package to declare subprograms for modifying the encapsulated state, protected objects, and arbitrarily complex data structures unless they fall into the categories above. The state of a partition (as far as rteplication is concerned) has been isolated in the data structures rooted in global variables in `Replicated` packages.

7 Non–Deterministic Replicas

The extension of the primary–backup scheme to non–deterministic partitions is still ongoing work. There are a couple of fundamental differences from the replication of deterministic partitions that complicate matters.

The basic idea is to emphasize the split of a partition into an especially distinguished replicated state (the `Replicated` packages), which must be deterministic, and the rest of the partition, which is now allowed to be non–deterministic. No specific task dispatching or queueing policies are required: as the state is accessible only through a well–defined, narrow interface, the replica manager can maintain its consistency.

As long as the distributed application contains only partitions that act either as a client or as a server, non–determinism causes only slight complications. Because re–executing a failed request might lead to different sequence of state changes, we must start a failed request from scratch, i.e., we must undo the effects of a previous attempt. Instead of collecting undo-information, it is sufficient that the backups do not update their state until the RPC request has been completely and successfully handled by the primary. At that moment, the backups can bring their states up to date by replaying the sequence of state accesses originally performed on the primary. If the primary could not handle the RPC request because it failed, one of the backups can re–execute the RPC from the initial state.

Unfortunately, the usual case are partitions that act as both client and server at the same time: in order to fulfill an RPC request, another — nested — remote call to some third partition is made. Since we have to restart a failed request from scratch, we have to undo all nested remote calls it made, too, otherwise the states of these other partitions might become inconsistent upon re–execution. This leads to a first, additional restriction: from within a replicated partition only remote calls to other replicated partitions are allowed. This makes sense insofar as the fault tolerance of a system composed of different replicated partitions is only as good as that of its weakest component, i.e. the partition that has the least degree of replication.

A further problem is the communication between concurrently executing remote calls. While the Ada standard basically allows an implementation of the PCS that serializes all RPCs (it only recommends allowing concurrent RPCs), such an implementation would be nearly useless. It is therefore necessary to deal with concurrently executing remote calls. If two such remote calls are independent, no special treatment is necessary: the two calls execute in parallel. The only way two concurrent remote calls may exchange information is by accessing the same protected object. At that moment, one RPC becomes a dependent of the other and it can only terminate successfully if the first one also terminates successfully: they have entered a conversation [10, 14]. These conversations are dynamic, for we cannot impose a limit upon the number of participating remote calls without giving up the goal of transparency.

The combination of dynamic conversations and nested RPCs can lead to a domino effect that can even leave the bounds of a single group. This can happen if a nested remote call enters a conversation with some other call, succeeds, and later its parent fails. In this case, the nested call must be undone, and therefore also its partners in the conversation. This problem can be alleviated somewhat — but not solved completely — by automatically including the parent of a nested remote call in any conversation the latter enters.

8 Future Work

Apart from the aforementioned difficulties with the replication of non–deterministic partitions, which still must be studied more closely, especially regarding ways to limit the domino effect, there are several other areas that merit further attention.

Recovery of failed replicas has not yet been considered. When a replica recovers and re–joins the group, it must receive the current state to properly initialize itself. One way to implement this state transfer would be to use the log of all previously executed state changes to bring the state of the new replica up to date, but (1) this log might become too long, and (2) it doesn't exist at all in active replication. A way to take and then transfer a snapshot of the state is therefore needed. It may be necessary to break the transparency rule and require some cooperation of the application to accomplish this: it may not always be possible to automatically collect the whole state, given that it may consist of arbitrarily complex, application–defined data structures.

Abortion of a remote call also must be studied more closely. What is to happen if an RPC to a replicated partition is to be aborted? Shall the replicated partition provide semantics closer to an atomic action (all–or–nothing property) or can it leave the responsibility for maintaining a consistent state in this case to the application? Both possibilities have their merits.

The configuration language must support the specification of replication and its characteristics. The configuration tool also must enforce and verify the various constraints placed on replicated partitions — mere assumptions are not acceptable from a software engineering perspective.

9 Acknowledgments

I thank the anonymous referees for their many useful comments on an earlier draft of this paper.

10 References

[1] Atkinson, C.; Moreton, T.; Natali, A.: *Ada for Distributed Systems*, The Ada Companion Series, Cambridge University Press 1988

[2] Birrell, A.D.; Nelson, B.J.: "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems* ***2****(1)*, pp. 39 - 59, 1984

[3] Chandra, T.D; Toueg, S.: "Unreliable Failure Detectors for Reliable Distributed Systems", *Technical Report TR95–1535*, Department of Computer Science, Cornell University, 1995

[4] Guerraoui, R.; Schiper, A.: "Fault Tolerance by Replication in Distributed Systems", *Proceedings of Ada-Europe '96*, Montreux, Switzerland, June 1996; published as *Lecture Notes in Computer Science* ***1088***, pp. 38 - 57, Springer 1996

[5] ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Intermetrics, Inc. 1995

[6] Jalote, P.: *Fault Tolerance in Distributed Systems*, Prentice Hall 1994

[7] Kermarrec, Y.; Pautet, L.; Tardieu, S.: "GARLIC: Generic Ada Reusable Library for Interpartition Communication", *Proceedings of TRI-Ada '95*, Anaheim CA, USA, pp. 263 - 269, Nov 1995

[8] Kermarrec, Y.; Pautet, L.: "Implementing recovery blocks in GNAT: a powerful fault tolerance mechanism and transaction support", *Proceedings of TRI-Ada '95*, Anaheim CA, USA, Nov 1995

[9] Malloth, C.; Felber, P.; Schiper, A.; Wilhelm, U.: "Phoenix: A Toolkit for Building Fault–Tolerant, Distributed Applications in Large Scale", *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio TX, USA, Oct 1995

[10] Randell, B.: "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering* ***SE-1****(2)*, pp. 220 - 232, 1975

[11] Schneider,F.B.: "Implementing Fault–Tolerant Services Using the State Machine Approach: A Tutorial", *ACM Computing Surveys* ***22****(4)*, pp. 299 - 319, Dec 1990

[12] Schonberg, E. et al.: "GNAT: the GNU – NYU Ada Translator, a compiler for everyone", *Proceedings of TRI–Ada '94*, Baltimore MD, USA, 1994

[13] Wellings, A.J.; Burns, A.: "Implementing Atomic Actions in Ada 95", *Technical Report YCS 263*, Department of Computer Science, University of York, UK, Jan 1996

[14] Xu, J.; Randell, B.; Romanovsky, A.B.; Rubira, C.M.F.; Stroud, R.J.; Wu, Z.: "Fault Tolerance in Concurrent Object–Oriented Software through Coordinated Error Recovery", *Proceedings of FTCS–25: Digest of Papers*, Pasadena CA, USA, 1995