Transparent Replication for Fault Tolerance in Distributed Ada 95

Thomas Wolf

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

Abstract

In this paper we present the foundations of RAPIDS ("Replicated Ada Partitions In Distributed Systems"), an implementation of the PCS supporting the transparent replication of partitions in distributed Ada 95 using semi–active replication. The inherently non–deterministic executions of multi–tasked partitions are modeled as piecewise deterministic histories. I discuss the validity and correctness of this model of computation and show how it can be used for efficient semi–active replication. The RAPIDS prototype ensures that replicas of a partition all go through the same history and are hence consistent.

Keywords: *Distributed Systems, Fault Tolerance, Group Communication, Piecewise Determinism, Semi–Active Replication.*

1 Introduction

Virtual nodes (i.e., partitions in Ada 95) in a distributed application can be rendered fault–tolerant by replication. A failure of a replica of a replicated partition can be masked thanks to the remaining replicas, which ensure that the partition remains available in spite of the failure.

Despite Ada's strong position in the development of dependable systems, the Ada 95 language standard addresses the issue of replication in distributed systems only in an "implementation permission" clause:

> *"An implementation may allow separate copies of a partition to be configured on different processing nodes, and to provide appropriate interactions between the copies to present a consistent state of the partition to other active partitions."*
>
> *— [ISO95, E.1(12)]*

With this statement, the standard argues in favor of a transparent solution to replication: as a distributed application is configured only *after* it has been programmed and compiled, it follows that the intent is that replication should be offered in a way transparent to the replicas of the replicated partition itself (*replica transparency*).

Note that although distribution is not quite transparent in Ada 95, this does *not* imply that replication shouldn't be either! The quoted paragraph also clearly addresses the issue of *replication transparency* by stating that a replicated partition should present a "consistent state" to other active partitions, which I take to mean that its behavior should be indistinguishable from that of a singleton partition. Replication is therefore also seen as transparent towards the other active partitions in the system.

In this paper, I present some results from my work on replication for fault tolerance in distributed Ada 95 [Wol98]. Replication in Ada 95 is complicated by the inherent non–determinism of partitions; a simple state machine approach [Sch90] is not applicable. I assume the following system model:

- A distributed Ada 95 application executes on an asynchronous distributed system (no timing assumptions, no synchronized clocks).
- The distributed system may be heterogeneous.
- Partitions do not share memory, i.e., there are no passive partitions in the system.
- Active partitions communicate by remote procedure calls through reliable channels over the network.
- Partitions are subject to crash failures only.
- Partitions are multi–tasked: incoming RPCs are handled concurrently in separate tasks.
- A view–synchronous group communication system provides consistent membership information and various reliable multicast primitives.
- Replicas are organized as a group.

The goal of replication is to render a partition fault–tolerant against crash failures in a way that ideally preserves both replica and replication transparency for the application.

The rest of this paper is structured as follows. I briefly review the various causes for non–determinism in Ada 95 in section 2. Section 3 presents the piecewise computation

Submitted for the 9^{th} *International Real–Time Ada Workshop (IRTAW–9)*, Tallahassee FL, USA, March 1999. To appear in *ACM Ada Letters*.

model, which abstracts from the vagaries of non–deterministic executions. In section 4, I give brief consideration to active and passive replication schemes, before describing in some detail semi–active replication in section 5. Section 6 gives a brief overview of Rapids, an implementation of a replication manager for the GNAT system using semi–active replication.

2 Non–Determinism in Ada 95

When a deterministic partition is to be replicated, active replication using the state machine approach of [Sch90] can be used. Replica consistency can be ensured by fulfilling the following two conditions:

- *Atomicity:* if one replica handles a request r, then all replicas do.
- *Order:* all replicas handle all requests in the same order.

This is insufficient if replicas are *non*–deterministic. In a multi–tasked partition for instance, task scheduling may well violate the order imposed on requests, leading to different state evolutions of replicas. Instead, the *sequence of state accesses* must by ordered identically on all replicas.

There are several causes of non–determinism in Ada 95. Besides the behavior of the tasking system, e.g. in its choices made for selective accept statements, explicit timing dependencies such as `delay` statements are the most obvious sources of non–determinism. Implicit timing dependencies also may cause non–deterministic behavior, e.g. the use of a pre–emptive time–sliced task scheduler or simply different message delivery times (even if their order is preserved) originating in network delays.

Because there is no coordinated time in a distributed Ada 95 application, these dependencies on time make the use of a deterministic task scheduler – i.e., one that always makes the same decisions at corresponding task dispatching points given the same set of tasks — impractical for guaranteeing replica determinism. Even if explicit time dependencies such as delays were forbidden, e.g. through pragma `Restrictions`, the implicit timing differences may cause replicas to diverge.

3 Piecewise Determinism

To overcome these difficulties, a more refined model of concurrent executions in Ada 95 is needed. The *piecewise deterministic computation model* [SY85, Eln93] views an execution as a sequence of deterministic state intervals that are separated by non–deterministically occurring events.

This model can be applied to Ada 95 as well. The signaling model of the language [ISO95, 9.10(3–10)] guarantees that all accesses to objects shared between any two tasks actually happen in some sequential order. As a result, shared objects must be protected using appropriate application–level synchronization through protected objects or rendezvous[1]. A partition is considered *erroneous* if it contains two tasks that access the same unprotected object without signaling in between.

An execution in the piecewise deterministic computation model is characterized by the events that occur. I distinguish two different classes of events.

Internal events account for non–determinism in the language model. They cover all task dispatching points, signaling actions, and events related to abort deferral:

- The choice made in a selective accept statement.
- The outcome of conditional and timed entry calls.
- Entering and leaving protected actions (locking/ unlocking protected objects).
- Queueing and requeueing on entry calls.
- Task creations, abortions, and terminations.
- Abortions in ATC.
- Initialization, finalization, and assignment of controlled objects.

External events basically account for non–determinism in the network:

- Delivery of an RPC request from another partition.
- Sending an RPC request to some other partition.
- Sending an RPC result back to the calling partition.
- Delivery of an RPC result from some other partition.

There are some additional internal events concerning local calls to subprograms whose results depend upon state outside the application, e.g. system calls like calling `Ada.Calendar.Clock`.

With the piecewise deterministic computation model, replicas can be synchronized by making sure that they all follow the same execution history, i.e. that they all go through the same sequence of events. Because signaling actions and task dispatching points are subsumed by the above list of events and any two concurrent state accesses must be separated by a signaling action, this will ensure that the sets of tasks and the states of all replicas are consistent.

Abortions and abort–deferred regions are included because otherwise abortions in ATC might basically cause replicas to diverge if abortions didn't happen at precisely the same logical moment on all replicas. Consider the example in fig. 1.

1. There are some special cases, though. A task could e.g. write a value to some unprotected variable and then create another task that read it. However, task creations also are signaling actions.

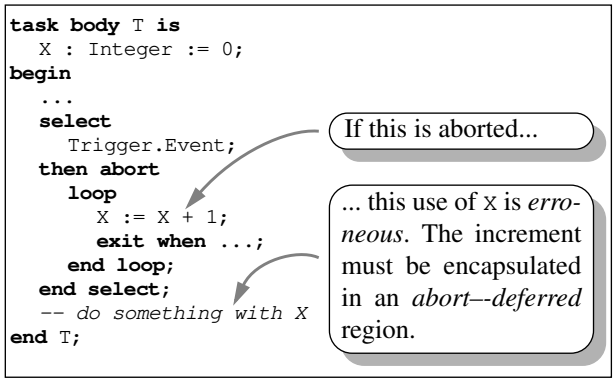

Fig. 1: Abortions in ATC

If the assignment in to variable `X` is aborted, `X` may be *abnormal* [ISO95, 9.8(21)] and hence its subsequent use is *erroneous* [ISO95, 13.9.1]. This is even true in centralized applications: even there, asynchronous aborts may cause problems for the application to make sure that state accessed in an asynchronous `select` statement remains consistent in the face of abortions. The critical sections of abortable parts (in the sense of making state modifications that may influence the further execution of the application) must therefore be encapsulated within abort–deferred regions. The language defines several constructs that defer abortion [ISO95, 9.8(6–11)], in particular, protected actions and also initialization, finalization, and assignment of controlled objects cannot be aborted. An abortion is delayed until the abort–deferred region has been completed. Because entering and leaving such abort–deferred regions is again subsumed by the above list of events, coordinating replicas by ensuring they follow the same sequence of events is sufficient to guarantee consistency. The semantics of Ada 95 will be preserved as abortions occur between the same two abort–deferred regions on all replicas.

4 Active and Passive Replication

Active replication is attractive because it offers a high availability of the replicated partition: as replicas execute in parallel, failures do not incur an additional overhead. It seems that one could use active replication with the piecewise deterministic computation model by reaching a consensus [BMD93] on events.

In fact, several systems employ this method, see e.g. SIFT [WLG+78], MAFT [KWFT88], or MARS [KFG+93]. However, these are special–purpose reliable hard real–time systems, not general systems. They are synchronous systems and use severely constrained tasking models, where inter–task communication is strongly reduced and task scheduling is restricted or even performed off–line, prior to run–time (static scheduling, e.g. in MARS). As a consequence, there are relatively few events, and thus the needs for interactive agreement are limited. Still, SIFT reports an overhead of up to 80% for replica synchronization [Pra96, p. 272] (admittedly assuming byzantine failures). MAFT achieves better performance by delegating the consensus protocols to dedicated hardware.

Nevertheless, this approach seems feasible for general systems, too, if one assumes deterministic task scheduling. This implies disallowing the use of any explicit time dependency (`delay` statements, calls to package `Ada.Calendar` or `Ada.Real_Time`). Under this assumption, progress can be measured by the task dispatching points passed. Consensus must then be reached only on the task dispatching point at which a message is delivered[2], as message deliveries are the sole remaining source of non–determinism. The most advanced task dispatching point must be taken as the common consensus value[3]. Those replicas that "lag behind" continue executing until this task dispatching point is reached and deliver the message only then. In this way, they all deliver all messages at the same task dispatching point and hence the sets of tasks continue to evolve identically on all replicas. The overhead of an additional consensus for each message delivery may be significant, but seems still tolerable.

However, this method violates replica transparency: forbidding the use of `delay` statements is a severe restriction and certainly not transparent. If delays are to be allowed, replicas must reach consensus on *each and every* event, not just on message deliveries! In a timed entry call for instance, all replicas must agree whether or not the entry call timed out, and if so, *when* this time–out occurred with respect to other task scheduling decisions taken during the delay. It is *not* sufficient if they agree only on the first condition (whether or not a time–out occurred), and they are thus forced to track and synchronize all task scheduling decisions, not just time–outs.

In other words, active replicas supporting the full tasking model of Ada 95 have to communicate over the network (run a consensus protocol) for each and every task scheduling decision. This seems impractical as it would tremendously slow down a replicated partition. In fact, one loses the main advantage of active replication, which is its high availability.

Passive replication, on the other hand, does not suffer from this communication overhead. As only the primary replica executes a request, no interactive agreement protocol is needed. However, in case of a failure, there's a higher latency until one of the backups has taken over. Furthermore, passive replication either requires checkpointing

2. Besides an earlier consensus needed for the totally ordered multicast necessary to ensure the ordering condition given in section 2.
3. Although all replicas make the same scheduling decisions, they do not execute in lockstep: one replica may already have advanced further than another.

coordinated with output[4], or remote calls must have the semantics of nested transactions. This latter approach [Wol97] cannot be implemented transparently because application–defined scheduling (through partial operations, i.e. entry calls) may conflict with the constraints imposed on scheduling by the serializability correctness criterion of transactions. Such conflicts may result in deadlocks that cannot be resolved in a transparent manner, and the transactional nature of remote calls would have to be exposed to the application [Wol98]. If transactions are to be offered in Ada 95, they must therefore be integrated at the language level[5].

5 Semi–Active Replication

Since both active and passive replication have their deficiencies when used for replication of non–deterministic partitions in Ada 95, I focussed on semi–active replication instead [BHV+90].

The piecewise deterministic computation model lends itself readily to this form of replica organization, which was pioneered in the Delta–4 project [Pow91]. Replicas are organized as a view–synchronous group [SS93], and all replicas execute incoming RPC requests. One replica is designated the *leader* and is responsible for taking all non–deterministic decisions. The other replicas — the *followers* — are then forced to make the same choices.

Replica synchronization can thus be achieved by logging events on the leader. The leader then synchronizes its followers using a FIFO–ordered reliable multicast [HT94]. The followers then replay the events as they occurred on the leader. As a result, all replicas will go through the same sequence of deterministic state intervals, which ensures replica consistency.

Because events, instead of message deliveries, are ordered and synchronized, semi–active replication has less stringent requirements on the group communication layer than active replication: other partitions may use a relatively simple reliable multicast for communication with the group of replicas instead of an expensive totally ordered multicast. (The FIFO multicast needed for synchronization within the group can be built easily upon the basic primitive of reliable multicast.)

Nevertheless, synchronizing the followers at each and every event would most probably be impractical: since internal events are bound to occur very frequently, this would entail a prohibitively high performance overhead as each task scheduling decision would again involve communication over the network for reaching agreement. Fortunately, this is not necessary. Synchronization is only needed at *observable events*:

- Sending an RPC result back to the client.
- Sending an RPC request to some other partition.

As long as the effects of state intervals remain purely local to the leader, the followers need not be informed of any events. The followers must be brought up to date only once the effects become visible to the rest of the system, i.e., when the leader sends a message beyond the group of replicas. Before doing so, the leader must update its followers in order to guarantee that they will reach the same state, otherwise, a failure might corrupt the overall consistency of the distributed application when one of the followers becomes the new leader.

Between observable events, the leader just logs events by buffering them in an event log. Just before it will execute an observable event, it multicasts this log to its followers. Only then may it proceed and perform the observable event. The log records an *extended state interval*, as it may contain many simple state intervals (or rather, the events delimiting them). Each observable event starts a new extended state interval. A follower recreates extended state intervals by replaying events from the logs it receives. When it has replayed all the events in the received logs, it just waits until the next extended state interval arrives from the leader, or until it becomes the leader itself due to a failure of the former leader.

A follower does not re–execute the observable event at the very beginning of an extended state interval as this would only result in a duplicate message being sent. It just uses the logged outcome of this event's execution on the leader to replay the event. The leader is thus the only replica that interacts with the rest of the system beyond the group of replicas.

While extended state intervals are generally started by observable events, the leader is free to synchronize the followers more tightly. This may be necessary when the event log buffer on the leader threatens to overflow: in this case, the leader has to send the log's current contents to the followers in order to make room for new events. Given a reasonably sized event buffer, the synchronization interval can still be kept large enough to obtain acceptable performance with this scheme.

Because followers do not participate in any interaction beyond the group, failures of followers are completely transparent with this replica organization. Upon a failure of the leader, however, one of the followers must take on the role of the new leader. It first replays any pending events it already had received from the failed leader to bring itself

4. Checkpointing a multi–tasked partition is not trivial: the complete tasking state with program counters, stacks, etc. must be included. Checkpointing is also limited to replicas running on homogeneous physical nodes.
5. And in this case, transactional RPCs would constitute an inversion of abstractions. In my opinion, transactions should be built upon RPCs, not the other way round!

up to date with the last known state of the latter. It then simply continues executing, henceforth logging events and synchronizing the remaining followers.

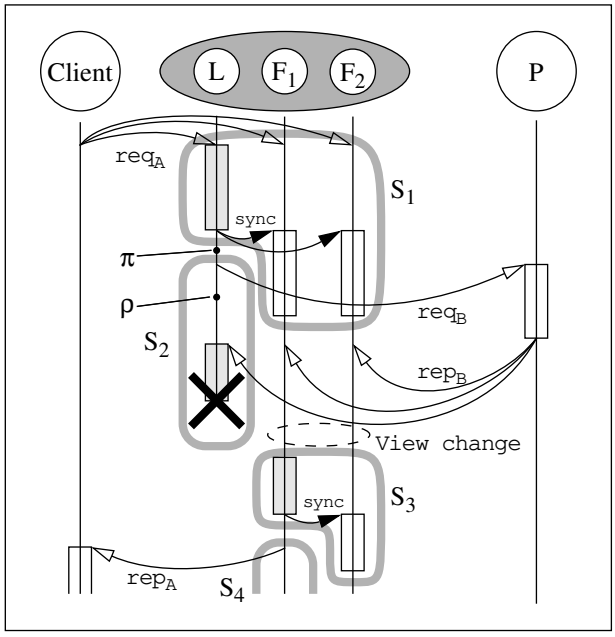

Fig. 2: A Failure in Semi–Active Replication

Fig. 2 shows a failure of the leader *L* of a replicated partition. It started executing a request $\texttt{req}_\texttt{A}$, made a nested remote call $\texttt{req}_\texttt{B}$ to some other partition, waited for the result and continued processing $\texttt{req}_\texttt{A}$ before failing. Just before $\texttt{req}_\texttt{B}$, it sent the events for the extended state interval S_1 to its followers, which then recreate this extended state interval by replaying the logged events.

When *L* finally fails, a view change occurs and follower F_1 is chosen as the new leader and continues executing from just after S_1. Note that the extended state interval S_3 may well be different from S_2, but since S_2 could not possibly have affected any other part of the system except *L* itself, the overall state of the system remains consistent.

If the failure and the view change had occurred before the reply $\texttt{rep}_\texttt{B}$ to the nested remote call had arrived, the new leader F_1 would have had no way to tell whether or not the former leader *L* did still send $\texttt{req}_\texttt{B}$. A failure at point π is indistinguishable from one at point ρ. Consequently, the new leader F_1 has no choice but (re–)execute the observable event $\texttt{req}_\texttt{B}$. If *L* had failed at point ρ (or later), this results in a duplicate request. An analogous situation arises when F_1 finally sends the reply $\texttt{rep}_\texttt{A}$ back to the client. If it fails after the synchronization of S_3, but before S_4 is synchronized, the then new leader F_2 again does not know whether or not $\texttt{rep}_\texttt{A}$ has been sent.

This implies that partitions must be able to deal with *duplicate messages*[6]. Messages must be tagged with a unique, system–wide identifier. Repeated RPC result messages are then simply ignored. Repeated invocations are more difficult to handle. A partition only handles the first RPC request message it receives. Later messages with the same message identifier are ignored if they arrive while the RPC is still in progress, or return the result of the first invocation if the RPC already has completed. The latter case necessitates that results of remote subprogram calls be retained, and therefore some kind of garbage collection of retained results must be provided (see section 6).

Semi–active replication based on the piecewise deterministic computation model can offer transparent replication of non–deterministic Ada 95 partitions. Replication transparency (i.e., transparency towards other partitions) is given at the application level, although partitions must be able to handle duplicate messages correctly at the system level. Yet the application level remains unaffected by this. Replica transparency, i.e. transparency towards the application level of the replicated partition itself also is maintained. The piecewise deterministic computation model supports the full tasking model of Ada 95. However, replica transparency in heterogeneous systems is only given as long as only failures are considered, and thus holds only for k–resiliency.

If recovery is to be included in the model, replica transparency cannot fully be maintained in the general case. If a new follower is to join a running group, it must get the group's current state. This *state transfer* can only be done transparently in a homogeneous system by taking a system–level checkpoint on one of the old group members and installing this checkpoint in the newly joining replica. If replicas execute on heterogeneous physical nodes, this state transfer is only possible for a restricted subset of all possible partitions and furthermore requires the cooperation of the application itself [Wol98]. In this case, replica transparency cannot be upheld totally.

6 RAPIDS

RAPIDS ("Replicated Ada Partitions In Distributed Systems") [Wol98] is an implementation of the semi–active replication scheme based on a piecewise deterministic computation model as presented above for the GNAT compiler. It is implemented within the run–time support and is thus largely transparent to the application.

The core of RAPIDS is the event log buffer, which is implemented within the PCS as a child package of Garlic [KPT95] called `System.Garlic.Rapids`. This choice

6. The reliable multicast primitive assumed for communication of clients and servers with a replicated partition (i.e., a group of replicas) already makes duplicate message detection necessary. However, here one needs a second duplicate message detection scheme at a higher level. It would be beneficial if this high–level duplicate message detection could exploit the fact that such a facility already exists in the (hidden) low–level protocols of the group communication layer.

was made because synchronization between the leader and its followers is triggered by observable events, i.e. sending messages, and these event occur within the PCS. Furthermore, the actual synchronization involves multicasting a message within the group, which can be done conveniently through a new protocol added to Garlic. This new protocol is only an interface to a third–party view–synchronous group communication system. Currently, RAPIDS uses the Phoenix [MFSW95] toolkit, but any other group communication system can be used as long as it satisfies view synchrony and offers a reliable multicast primitive.

The PCS has also been modified to include unique message identifiers in all messages. Using these, duplicate message detection is implemented. Garbage collection of retained results has also been included in Garlic. Whenever a partition A sends a message to another partition B, it piggybacks information about the messages it already received from B. Partition B can then discard these messages.

RAPIDS actually offers three different interfaces for logging and replaying events and for transferring the event log from the leader to the followers. The PCS uses direct calls to a first interface given by `System.Garlic.Rapids` to handle external events. To handle internal events, the tasking support GNARL has been modified to use a callback interface to RAPIDS for logging and replaying events for all task scheduling decisions. Some other packages of the run–time support also use callbacks to RAPIDS, e.g. `System.Finalization_Implementation` uses this to log and replay events regarding entry and exit of the abort–deferred regions given by initialization and finalization of controlled objects. Finally, there is a public interface to the event log in `System.RPC.Replication` (shown in fig. 3) for use by the standard libraries, which also may have to log and replay events, e.g. `Ada.Calendar.Clock` of file accesses.

```
package System.RPC.Replication is
   type Event is abstract tagged private;

   procedure Log (E     : in     Event'Class);

   procedure Get (E     : in out Event'Class;
                  Valid :    out Boolean);

   procedure Remove;

   procedure Send_Log;

private
   ...
end System.RPC.Replication;
```

Fig. 3: Public Interface of the Event Log

The event log buffer is organized as a heterogeneous FIFO list, storing event descriptors derived from the abstract tagged type `Event`. A leader can append events to the log using the `Log` subprogram, and can multicast its log to its followers using the `Send_Log` operation, which also empties the log. Each event also contains a unique *group–wide task identifier* for the task involved. A follower replays events in the order they have been logged. The `Get` subprogram blocks the calling task until an event matching the tag of the actual parameter `E` *and* that tasks's group–wide ID is frontmost in the log. It then returns the event with `Valid` set to `True`. (On a leader, `Get` returns immediately with `Valid` set to `False`.) With the `Remove` operation, a follower can actually remove the frontmost event from the log once it has replayed the event. This makes the next event in the log become the new frontmost event, and thus some other call to `Get` may be unblocked.

If the log on a follower is empty, i.e., has been wholly replayed, `Get` also blocks until either the next extended state interval is received from the leader and a matching event appears at the front of the log, or the follower becomes a leader when the old leader has failed. In this case, `Get` returns even if no matching event has appeared (once the log has been exhausted, cf. section 5) with `Valid` set to `False`: as the former follower is now a leader, it may make its own choices.

With this interface, event logging and replay can be implemented following the pattern shown in fig. 4.

```
with System.RPC.Replication;
package Example is

   package Repl renames System.RPC.Replication;

   type Event is new Repl.Event with
      record
         -- The characterizing data of the event.
      end record;
   for Event'External_Tag use "Example.Event";

   procedure The_Operation (...) is
      The_Event   : Event;
      Is_Follower : Boolean;
   begin
      Repl.Get (The_Event, Is_Follower);
      if Is_Follower then
         -- Use the description in 'The_Event' to
         -- replay the event. Then remove it:
         Repl.Remove;
      else -- Leader!
         -- If it's an observable event, send the
         -- log to the followers.
         Repl.Send_Log; -- Only if observable!
         -- Do the event.
         Do_Operation;
         -- Log the event.
         The_Event := (Repl.Event with ...);
         Repl.Log (The_Event);
      end if;
   end The_Operation;

end Example;
```

Fig. 4: Pattern for Event Logging and Replay

Event replay on a follower is *atomic*: other tasks that also might call `Get` for other events will remain blocked until this event is removed. On the leader, event logging must be done in the right places. `Do_Operation` itself should not cause additional non–deterministic events. (If it did, `Do_Operation` instead of `The_Operation` should be implemented using this pattern.) Also, the event must be logged in the right moment. Consider for example the event of locking a protected object. Obviously, a task must *first* get the lock and *then* log the event for it; if it logged the event first, some other task might actually get the lock first, and the event log would be inconsistent with the actual execution history.

Group–wide task identifiers are implemented through task creation events. These events contain both the group–wide ID of the creating task and that of the newly created task. When a follower replays the event, it therefore also knows which group–wide ID it must assign to the new task.

References to time such as `delay` statements or calls to `Ada.Calendar.Clock` all generate events. RAPIDS implements a mapping of time values such that all replicas always run logically on the initial leader's time base. This avoids that time suddenly jumps backwards when a failure occurs and thus guarantees the monotonicity of time for the application.

Not all internal events are logged by RAPIDS. It makes a distinction between system tasks, which are local to the run–time support, and application tasks. Only events involving application tasks are logged and replayed. The system tasks, however, execute independently on all replicas: they have to, because the run–time support must do different things on the leader than on a follower. At the interface of the PCS, tasks may change their status: an application task calling e.g. `System.RPC.Do_RPC` becomes a system task inside that call, and conversely a system task becomes an application task for the time it executes a remotely called subprogram.

RAPIDS is currently (Nov 1998) still in a prototype stage. It still needs serious optimization efforts, and it doesn't yet handle dynamically bound remote calls through remote access–to–subprogram or remote access–to–class–wide values. Also, events due to assignments of controlled objects are not yet handled, as this seems to require some cooperation of the compiler's part.

7 Conclusion

Modeling executions of Ada 95 partitions using a piecewise deterministic computation model overcomes the problems due to non–determinism that occur in replication. The model, together with the abstractions in the language standard, abstracts from timing dependencies and thus makes replication possible. It seems that semi–active replication is the most appropriate replication scheme for general Ada 95 partitions using the full tasking model of the language.

A prototype of a replication manager called RAPIDS ("Replicated Ada Partitions in Distributed Systems") has been developed. It guarantees replica consistency by logging non–deterministic events on the leader and replaying them on the followers. Although this project is still in an early prototype stage, first results are encouraging, indicating that efficient replication is attainable using this model.

8 References

[BHV+90] Barrett, P. A.; Hilborne, A. M.; Veríssimo, P. et al.: "The Delta–4 Extra Performance Architecture XPA", in *Proceedings of the 20th International Symposium on Fault–Tolerant Computing Systems (FTCS–20)*, pp. 481 – 488, Newcastle upon Tyne, UK, June 1990.

[BMD93] Barborak, M.; Malek, M.; Dahbura, A.: "The Consensus Problem in Fault–Tolerant Computing", *ACM Computing Surveys **25**(2)*, June 1993, pp. 171 – 220.

[Eln93] Elnozahy, E. N.: *Manetho: Fault Tolerance in Distributed Systems using Rollback Recovery and Process Replication*, PhD Thesis, Rice University, Houston TX, USA, Oct. 1993.

[HT94] Hadzilacos, V.; Toueg, S.: "A Modular Approach to Fault–Tolerant Broadcasts and Related Problems". *Technical Report TR94–1425*, Department of Computer Science, Cornell University, Ithaca NY, USA, 1994.

[ISO95] ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Lecture Notes in Computer Science **1246**, Springer Verlag, 1997; ISO, 1995.

[KFG+93] Kopetz, H.; Fohler, G.; Grünsteidl, G. et al.: "Real–Time Systems Development: The Programming Model of MARS", in *Proceedings of the International Symposium on Autonomous Decentralized Systems*, pp. 190 – 199, Kawasaki, Japan, Mar. 1993.

[KPT95] Kermarrec, Y.; Pautet, L.; Tardieu, S.: "GARLIC: Generic Ada Reusable Library for Interpartition Communication", in *Proceedings of Tri-Ada '95*, pp. 263 – 269, Anaheim CA, USA, Nov. 1995.

[KWFT88] Kieckhafer, R. M.; Walter, C. J.; Finn, A. M.; Thambidural, P. M.: "The MAFT Architecture for Distributed Fault Tolerance", *IEEE Transactions on Computers **37**(4)*, Apr. 1988, pp. 398 – 405.

[MFSW95] Malloth, C.; Felber, P.; Schiper, A.; Wilhelm, U.: "Phoenix: A Toolkit for Building Fault–

Tolerant, Distributed Applications in Large–Scale Networks", in *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio TX, USA, Oct. 1995.

[Pow91] Powell, D. (Ed.): *Delta–4: A Generic Architecture for Dependable Distributed Computing*, volume 1 of *ESPRIT Research Reports, Project 818/2252 — Delta–4*, Springer Verlag, 1991.

[Pra96] Pradhan, D. K.: *Fault–Tolerant Computer System Design*, Prentice Hall, 1996.

[Sch90] Schneider, F. B.: "Implementing Fault–Tolerant Services using the State Machine Approach", *ACM Computing Surveys **22**(4)*, Dec. 1990, pp. 299 – 319.

[SS93] Schiper, A.; Sandoz, A.: "Understanding the power of the virtually–synchronous model". *Technical Report BROADCAST TR93–04*, ESPRIT Basic Research Project 6360, 1993.

[SY85] Strom, R. E.; Yemini, S.: "Optimistic Recovery in Distributed Systems", *ACM Transactions on Computer Systems **3**(3)*, Aug. 1985, pp. 204 – 226.

[WLG+78] Wensley, J. H.; Lamport, L.; Goldberg, J. et al.: "SIFT: Design and Analysis of a Fault–Tolerant Computer for Aircraft Control", *Proceedings of the IEEE **66**(10)*, Oct. 1978, pp. 1240 – 1255.

[Wol97] Wolf, T.: "Fault Tolerance in Distributed Ada 95", in *Proceedings of the 8th International Real–Time Ada Workshop*, Ravenscar, UK, Apr. 1997, ACM Ada Letters **XVII**(5), pp. 106 – 110.

[Wol98] Wolf, T.: *Replication of Non–Deterministic Objects*, PhD Thesis #1903, Swiss Federal Institute of Technology, Lausanne, Switzerland, Nov. 1998.