Fault Tolerance by Transparent Replication for Distributed Ada 95

Thomas Wolf and Alfred Strohmeier

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne Ecublens
Switzerland

e–mail: {Thomas.Wolf, Alfred.Strohmeier}@epfl.ch
phone: +41 (21) 693.42.37 — fax: +41 (21) 693.50.79

Abstract. This paper presents the foundations of RAPIDS (“Replicated Ada Partitions In Distributed Systems”), an implementation of the Distributed Systems Annex E incorporating the transparent replication of partitions in distributed Ada 95 applications. RAPIDS is a replication manager for semi–active replication based on a piecewise deterministic computation model. It guarantees replica consistency for arbitrary Ada 95 partitions despite the inherent non–determinism of multitasking, offering k–resilient partitions. The RAPIDS prototype is implemented for the GNAT compiler and evolved from its PCS, Garlic.

Keywords. Distributed Systems, Fault Tolerance, Non–Determinism, Piecewise Determinism, Replica Consistency, Semi–Active Replication.

1 Introduction

Measures for fault tolerance always introduce redundancy in the system. Examples are data redundancy (e.g., error–correcting codes), time redundancy (re–execution after a failure), or space redundancy (replication). Replication in particular is well suited for rendering nodes in a distributed application fault–tolerant, the idea being that if one replica of a node fails, the remaining replicas can continue to provide the node’s services. The goal of replication for fault tolerance is thus the transparent masking of failures.

The main difficulty of replication of virtual nodes is maintaining the consistency of all the replicas. In Ada 95, this is further complicated by the inherent non–determinism. In section 2, we present our system model, before we examine the issue of non–determinism in detail in section 3. We show that this non–determinism cannot be exorcized even assuming that tasking was deterministic. In section 4, we briefly discuss our earlier attempt at replication in Ada 95 [Wol97] and show that it is inadequate for solving the problem of replication of non–deterministic partitions.

In section 5, we present a model of computation based on piecewise determinism [SY85, Eln93] that allows us to overcome this problem. Under the piecewise computa-

Submitted to *Ada Europe ’99*, Santander, Spain, June 1999. To appear in the conference proceedings in *Lecture Notes in Computer Science*, Springer Verlag.

tion model, replica consistency can be ensured by coordinating the replicas using semi–active replication, which was pioneered in the Delta–4 project [Pow91]. In this scheme, all non–deterministic decisions are taken on one distinguished replica; all other replicas are forced to adopt that choice.

Section 6 gives a very brief overview of the implementation of RAPIDS [Wol98], a replica manager for transparent semi–active replication based on piecewise determinism for the GNAT compiler. We outline the basic structure of RAPIDS and indicate its interactions with other parts of the run–time support. A detailed presentation of RAPIDS' implementation is intended for a later paper.

The conclusion summarizes the main findings of our work and reports on RAPIDS' current state.

2 The System Model

A distributed application in Ada 95 is composed of a set of virtual nodes (called "partitions") that themselves are indivisible as far as distribution is concerned and that execute on various physical nodes (computers), which may be heterogeneous [ISO95, E(6)]. The physical nodes are connected by a network, over which they communicate with each other. A partition in Ada 95 is a collection of library units, some of which constitute its interface towards the other partitions of the application. These well–defined interfaces are given by the specifications of certain library units that have to be marked in the source using special categorization pragmas.

Ada 95 distinguishes between active and passive partitions. Passive partitions are intended to model memory shared between virtual nodes, either physically or through a virtual distributed shared memory system, and have no thread of control associated with them. Their interface may contain remotely accessible data objects, which can be accessed by other (active) partitions. Active partitions *do* have a thread of control. Different active partitions communicate with each other only through their interface library units by means of remote procedure calls (RPCs, [BN84]). The language standard also defines ways for making dynamically bound remote calls through the use of remote access–to–procedure or remote access–to–class–wide types, but these too boil down ultimately to RPCs. Direct remote data access between active partitions does not exist in Ada 95.

Remote procedure calls are a communication abstraction built on top of simple message passing. They have "at–most–once" semantics. In Ada 95, the type safety is maintained even across RPCs because the compiler knows which subprograms are remotely callable (through the categorization pragmas) and hence can apply all the semantic checks to ensure the integrity of the distributed application, just like for local subprogram calls. The run–time support implements the message protocols underlying the RPC abstraction transparently in its Partition Communication Subsystem (PCS). The language standard strongly suggests [ISO95, E.5(24,25,28)] that a partition should handle incoming RPCs concurrently, each one in its own task.

Given this model defined by the language standard, we examine *transparent replication* for fault–tolerance assuming an asynchronous *heterogeneous* distributed system. The partitions of a distributed application do not share memory, but communicate only by RPC over the network through *reliable channels*[1]. Partitions are assumed to be subject to *crash failures* only (fail–silent behavior). We also assume that we have a *view–synchronous* group communication system [Bir93] at our disposal, which supplies the group abstraction, i.e. consistent membership information and a reliable multicast primitive.

3 Non–Determinism in Ada 95

When the replicas of a partition are deterministic, the state machine approach of [Sch90] can be employed for active replication. Consistency between the replicas can then be ensured by satisfying two conditions:

- *Atomicity:* if one replica handles a request *r*, then all replicas do.
- *Order:* if one replica handles a request r_2 after having handled a request r_1, then all replicas handle r_2 only once they have handled r_1.

This guarantees that all replicas handle the same requests in the same order, and — since they are assumed to be deterministic — their states will evolve in a consistent way.

With *non*–deterministic replicas request ordering is not sufficient. Partitions in Ada 95 generally are multitasked, and task scheduling may well violate the order imposed on request handling, leading to diverging evolutions of replicas. Instead of ordering the delivery of requests, the state accesses performed by the requests must be ordered: all replicas must go through a *common sequence of state accesses*.

The non–determinism perceived at the language level in Ada 95 has several causes. The most obvious ones are all related to the behavior of the tasking system, for instance in the choice made in a selective accept statement. Explicit dependencies on time (`delay` statements) also are a source of non–determinism, in particular when used in the context of asynchronous transfers of control or timed entry calls. Because there is no global notion of time in a distributed Ada 95 application and because the physical nodes the replicas of a partition execute on may run at different speeds, timed entry calls may time out on one replica whilst succeeding on another.

Implicit timing dependencies — introduced e.g. by the use of time–sliced pre–emptive task scheduling, or by varying message delivery delays originating in the communication layer — may cause executions on different replicas of a partition to diverge.

Even if one forbade the use of delays (e.g. through pragma `Restrictions`) in a partition that was to be replicated, it is *not* possible to side–step the problem of non–deter-

1. A message sent on a reliable channel will eventually be received by the recipient if both the sender and the receiver do not fail; if link failures are repaired eventually, this property can be implemented though message retransmissions.

minism simply by using deterministic task scheduling on all replicas, i.e., a scheduler that is guaranteed to make the same decisions on all replicas given the same sets of tasks. The problem is illustrated by fig. 1.

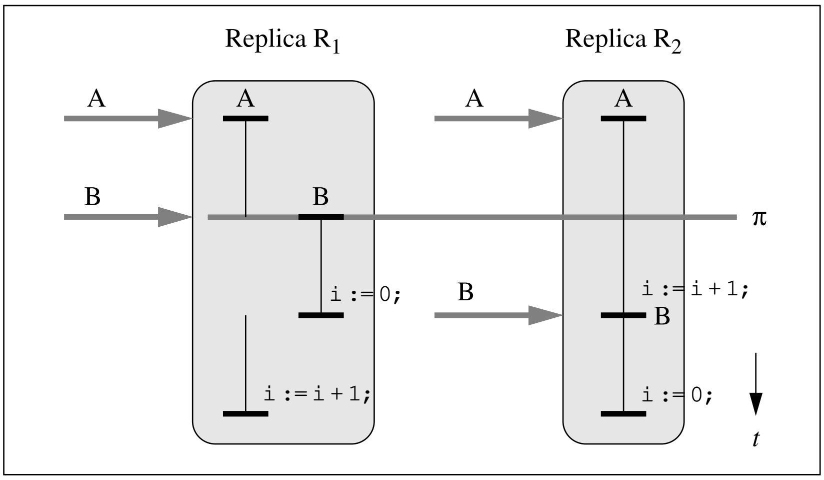

Fig. 1 Non–Determinism due to Delivery Delays

In this figure, two replicas of a multitasked partition both deliver two RPC requests A and B in the same order. Despite their using a deterministic task scheduler, their states diverge. At point π in fig. 1, the scheduler on replica R_1 has two tasks to schedule and decides to let B execute. On replica R_2, there is only the task for A at point π because the request B didn't arrive yet. As a result, R_2's scheduling differs from that on R_1, which may lead to state inconsistencies.

4 Transactions

One approach to synchronize the order of state accesses — short of executing all RPC requests strictly sequentially, which is not a viable option — would be based on the use of transactions.

We assume that state accesses are identified somehow, for instance through a new pragma `Replicated` the application developer uses to designate all global objects that are to be considered "state" [Wol97]. Instead of active replication, we use a passive replication scheme known as coordinator–cohort replication. One replica is singled out as the coordinator: it is the only one to handle requests. On the coordinator, the sequence of state accesses is logged. When a request is completed, this sequence of state accesses is multicast to all cohorts, which then apply the state changes in the order recorded.

Upon a failure of the coordinator, one of the cohorts becomes the new coordinator. It has to restart requests from scratch as we cannot make the new coordinator continue at precisely the same point in the flow of control where the former coordinator failed[2]. Due to the non–determinism, re–execution might follow a different path in the control flow than the original execution on the former coordinator. If the former coordinator had made nested remote calls to yet other partitions, it is therefore not guaranteed that the new coordinator will redo them with the same parameters or that it will redo them at all. As a consequence, these nested remote calls must be undone before the new coordinator takes over, otherwise the "at–most–once" semantics of RPCs might be violated and the overall consistency of the distributed application would be compromised. The destination partitions of these nested calls must therefore *roll back*. Rollback of nested calls may lead to a domino–effect, as such a nested call might have communicated with yet other RPCs from other partitions by exchanging data through global protected objects. To avoid this domino effect, RPCs must be *isolated* from each other. As a result, RPCs must be implemented *transparently* as *nested transactions* [Mos81].

This approach is rather heavy–weight, and its disadvantages are discussed at length in [Wol98]. It burdens the compiler not only with translating all accesses to objects marked by the pragma `Replicated` to include logging, but also requires it to rewrite these objects automatically to include recoverability and transactional concurrency control (e.g., to include locks if strict two–phase locking is used for concurrency control). Moreover, the effects of a failure of the coordinator of a replicated partition are not local to the group of replicas: because nested remote calls made from a replicated partition have to be rolled back, even non–replicated partitions would have to implement RPCs as nested transactions.

The most serious problem with this idea of transparently implementing RPCs as nested transactions are deadlocks, though. As shown in fig. 2, the interactions between application–level scheduling implemented through the entry queues of protected objects and the constraints imposed by the serialization of transactions may lead to deadlocks that cannot be resolved transparently. We will show this assuming the use of strict two–phase locking for concurrency control; a similar argument can be made for concurrency control based on timestamp ordering (see [Wol98]).

In fig. 2, two concurrent RPCs, i.e., nested transactions, access two global protected objects x_1 and x_2. Both transactions first make a write access to one of the objects and then try to make an entry call on the other object. We assume that the barriers of these entry calls are such that they are open only when the write access of the other transaction occurred first. With the interleaving of operations shown in fig. 2, both transaction first execute their write accesses. Transaction t_1 then makes its entry call on object x_2 and is blocked despite the barrier being open because the object is locked by t_2. The same happens with transaction t_2 and object x_1. Suppose now that t_1 is aborted in an attempt to break the deadlock. At that moment, t_2 could basically proceed with its entry call as t_1 no longer holds the lock of object x_2. However, t_2 will remain blocked

2. In a heterogeneous system. In a homogeneous system, one could basically take a checkpoint including the program counter, tasking state, stacks, etc. However, taking such a checkpoint is far from evident in a multitasked environment, and even if it can be done, it might prove too costly to do each time the coordinator sends a message to some other partition.

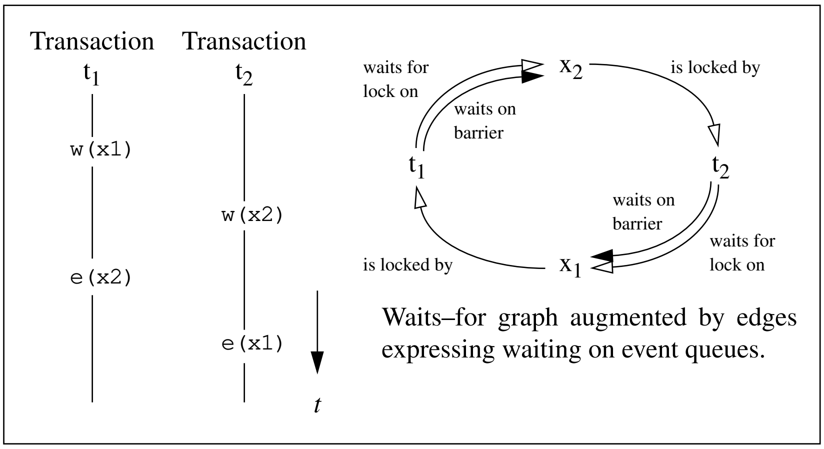

Fig. 2 An Unresolvable Deadlock

because now the barrier is closed! (Recall that the assumption is that t_1's write access opens the barrier, but this write access has just been undone.)

As a result, this not at all uncommon case that works perfectly well in normal Ada 95 will always deadlock when the partition is replicated! Furthermore, the deadlock *cannot be broken*: one transaction will always be blocked by the two–phase locking rules while the other one will block on the application–defined entry barrier! The application must therefore be aware of this possibility and be written in such a way that these unbreakable deadlocks do not occur. Transactions cannot be used *transparently* to ensure consistency in a distributed application: they must be made visible to the application. How to best integrate transactions at the application level in Ada 95 is certainly an area worth exploring, but runs counter to our goal of offering fault tolerance by replication in a *transparent* manner.

5 Piecewise Determinism

We have therefore focused our attention on a more refined modeling of concurrent executions in Ada 95. In the piecewise deterministic computation model [SY85, Eln93], an execution is seen as a sequence of *deterministic state intervals* separated by *non–deterministically occurring events*.

This describes precisely executions in Ada 95. The language abstractions guarantee that any execution history in Ada 95 is linearizable [HW90]. The signaling model of Ada 95 [ISO95, 9.10(3–10)] requires that all concurrent accesses to an object be in fact sequential in some order, i.e., actions of different tasks on one and the same object are only allowed if the action of one task signals the action of the other task. This basically means that all global state that is to be accessed concurrently must be encapsulated in protected objects, or be protected by appropriate application–level synchronization using rendezvous, otherwise the partition is erroneous!

Without loss of generality, we assume that any global data is encapsulated in protected objects[3]. Because protected objects implement linearizability trivially through mutual exclusion, and because linearizability is a local property (i.e., the composition of individual linearizable execution histories of different objects yields again a linearizable history), the execution of the whole partition is linearizable. Any interleaving of piecewise deterministic state intervals that will occur in a correct Ada 95 partition will yield a global linearizable history.

5.1 Events

An execution history under piecewise determinism is characterized by the sequence of events that occur: the events describe the sequence of deterministic state intervals. We distinguish external and internal events.

External events occur at any interaction of a partition with the rest of the system and account for non–determinism in the communication support (cf. fig. 1 in section 3). They are:

- Delivery of an RPC request message sent by some other partition.
- Sending an RPC answer message back to the calling partition to return the results of a remote call.
- Sending an RPC request message to some other partition.
- Delivery of an RPC answer message from some other partition.

Internal events are all sources of non–determinism within a partition. They basically account for non–determinism in the tasking system:

- The decision, which entry in a `select` statement is accepted.
- The outcome of timed and conditional entry calls.
- Entering and leaving protected actions, in particular locking and unlocking protected objects.
- Queueing and requeueing on entry calls.
- The creation, termination, and abortion of tasks.
- Abortion of an abortable part of an asynchronous `select` statement.
- Initialization, finalizations, and assignments of controlled objects.

A special kind of internal events are local calls to subprograms whose results depend upon state outside the application, e.g. system calls. An example is the `Clock` function in the standard package `Ada.Calendar`, which returns an approximation of "wall clock" time. Such calls (and their results) also must be considered internal events.

3. The case where a protected object acts as a lock for some unprotected data is functionally equivalent; and synchronization at the application level through rendezvous also can be modeled by protected objects.

5.2 Semi–Active Replication

The piecewise deterministic model lends itself well to semi–active replication [BHV+90] for coordinating the replicas. Semi–active replication has been introduced in the Delta–4 project [Pow91]. Replicas are organized as members of a group, and all replicas execute incoming RPC requests. One replica is designated the *leader*: it is responsible for taking all non–deterministic decisions. These decisions are propagated to the other replicas — the *followers* — that then are forced to make the same choices.

Replica consistency with semi–active replication can thus be achieved by logging events on the leader. The leader then synchronizes its followers by sending them the events using a FIFO–ordered reliable multicast. The followers then replay the events as they occurred on the leader. As a result, all replicas will go through the same execution history, i.e, sequence of deterministic state intervals, and consequently state accesses will occur in the same order on all replicas.

5.3 Correctness

The correctness of this approach relies on the semantics of the programming language Ada 95. The signaling model guarantees that all state accesses are indeed separated by events and happen in different deterministic state intervals, because all signaling actions defined in the language standard are subsumed by the set of internal events. Thus, ordering state intervals (by ordering events) also orders concurrent accesses to objects.

Abortions in asynchronous select statements might basically cause the replicas' states to diverge if the abortion didn't happen at precisely the same logical moment during execution. Consider the example in fig. 3 below.

```
task body T is
   X : Integer := 0;
begin
   ...
   select
      Trigger.Event;
   then abort
      loop
         X := X + 1;
         exit when ...;
      end loop;
   end select;
   -- do something with X
end T;
```

If this assignment is aborted...

... then this use of `X` is *erroneous*, because `X` might be *abnormal*. The increment of `X` must be encapsulated within an *abort–deferred* region to correct this code snippet.

Fig. 3 Aborts in asynchronous `select` statements

The abortable part of this asynchronous `select` statement is started if the entry call on the protected object `Trigger` is not selected immediately or is requeued. If the triggering statement completes, the abortable part is aborted (provided it had been started and didn't complete already). This abortion might happen basically at any time, in the case of replication even at different times during the execution of the abortable part on any two replicas. Such asynchronous aborts may be problematic even in a centralized application, as even then it is nearly impossible to make sure the state accessed by the abortable part remains consistent in the face of abortion. Furthermore, if an assignment (other than to a controlled object, see below) is disrupted by the abortion, the target object of the assignment may become *abnormal* [ISO95, 9.8(21)], and any uses of values of abnormal objects are considered *erroneous* in the language standard [ISO95, 13.9.1]. The critical sections of abortable parts (in the sense of making modifications that might have an influence on the further execution) must therefore be implemented as *abort–deferred regions*. The language defines several constructs that defer abortion [ISO95, 9.8(6–11)], in particular, protected actions as well as initialization, finalization, and assignment of controlled objects cannot be aborted. The abortion takes place only once the abort–deferred construct is completed. The abort–deferred regions and also the possible triggers of such asynchronous aborts again are all subsumed by the above list of internal events. By replaying the event log on the followers, it is therefore guaranteed that an abort happens between the same two abort–deferred regions as on the leader, which is sufficient to maintain the semantics of Ada 95 given that any critical state modification in an abortable part is done within an abort–deferred region.

5.4 Replica Synchronization: Observable Events

Synchronizing the followers at each and every event that occurs on the leader would most probably impractical: it would incur a prohibitively high performance overhead as internal events in particular are bound to occur very frequently. Fortunately, this is not necessary. 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 when and if the effects become observable to the rest of the system. This leads to the notion of *observable events*: if the leader sends information to any other partition in the system (or outputs something), this data depends upon the precise sequence of state intervals executed, and followers must be guaranteed to go through the same sequence of state intervals in order to produce the same observable event. All other events have purely local effects — should the leader fail, the rest of the system is still in the same state as if they hadn't happened at all, and a follower, once it has become the new leader, may make different choices without disturbing the overall consistency of the application. Synchronization is therefore needed only before an observable event occurs. An observable event is one of the following:

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

Between observable events, the leader logs any occurrences of external and internal events by buffering them in an event log. Just before an observable event is about to happen, the leader multicasts this event log to the followers. Only then it may proceed and perform the observable event. We call the path of execution described by the event log an *extended state interval* as it my contain many simple state intervals together with the events relating them. Each observable event starts a new extended state interval. When a follower receives an event log from the leader, it proceeds with the execution, replaying the logged event outcomes and thereby recreating the extended state interval. It does not replay the initial observable event, though: since the leader already executed it, doing so would only result in a duplicate invocation. Instead, the followers will just use the logged outcome of the event. The leader is thus the only replica that ever interacts with the rest of the system. When an event occurs that is not in the log, this signifies that the whole log has been replayed, and the follower blocks until it receives the next event log from the leader, or until it becomes the new leader itself due to a failure of the former leader.

In general, the leader starts a new extended state interval each time an observable event occurs. But it is free to define additional points in the execution that also start new extended state intervals, i.e., that also make it send its event log to the followers. For instance, the leader can start a new extended state interval whenever its event log buffer threatens to overflow. This does not change the correctness of the approach, it just leads to a closer synchronization between the leader and the followers.

5.5 Failures

The failure of a follower is completely transparent. Because a follower never interacts with the rest of the system beyond the group of replicas, its failure cannot possibly have any effects on the distributed application. When a follower fails, it is simply excluded from the group.

Upon a failure of the leader, one of the followers must become the new leader. It then resumes execution at the point the former leader had last communicated to the followers. It first re–executes any pending events in the event log. Once the event log is exhausted, the new leader simply continues executing, henceforth logging events as they occur and sending its event log to the remaining followers before each observable event.

Although the old leader may have progressed since it had sent its event log for the last time before its failure, the rule that synchronization must take place before any observable event ensures that any actions of the failed leader could not affect the system. This is similar to write–ahead logging in databases, where all necessary information for undoing or redoing an action must be in the log before the state may be updated. Here, the replicas must have received the leader's decisions that led to an observable event before the leader may execute this event. This is illustrated in fig. 4.

The leader L of a replicated partition started executing a request $\texttt{req}_{\mathrm{A}}$, made a nested remote call $\texttt{req}_{\mathrm{B}}$ to some other partition P, waited for the result, and continued processing $\texttt{req}_{\mathrm{A}}$ before it finally failed. Just before it performed $\texttt{req}_{\mathrm{B}}$, it sent its event log for the extended state interval S_1 to its followers F_1 and F_2, which replay the log to execute S_1. When L fails, a view change occurs and follower F_1 becomes the new leader and continues execution from just after S_1. Note that the extended state interval S_3 may well be different from S_2 (except for the initial observable event $\texttt{req}_{\mathrm{B}}$, which is

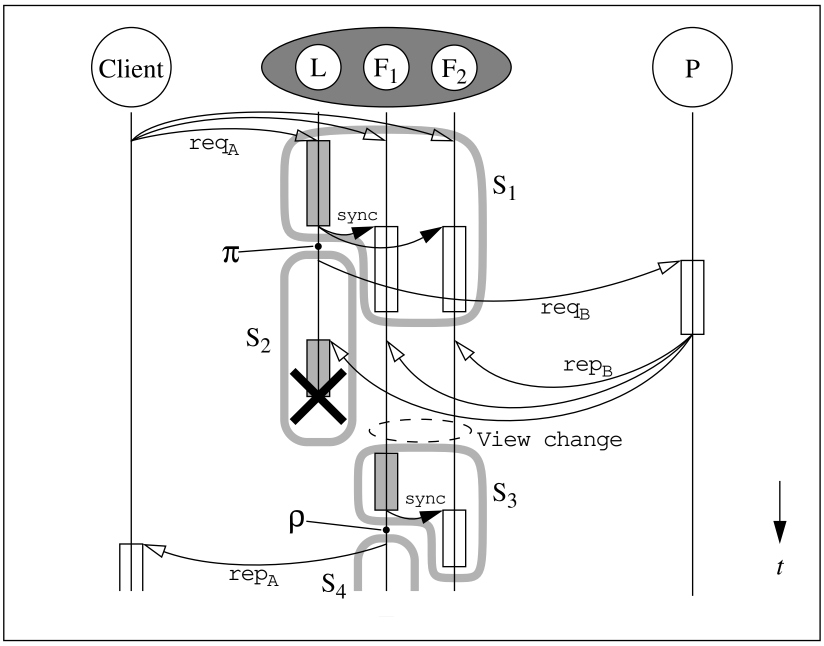

Fig. 4 Failures in Semi–Active Replication

guaranteed to be identical), but since S_2 could not possibly have affected any other part of the system except *L* itself, the application's overall state is still consistent.

Note that in the example in fig. 4, we assume that communication with a group of replicas is always by reliable multicast: even the reply $\texttt{rep}_B$ is multicast from *P* to all replicas. F_1 therefore doesn't need to re–execute $\texttt{req}_B$ at the very beginning of S_3 if the answer already arrived. If the view change due to the failure of *L* had occurred *before* $\texttt{rep}_B$ had arrived, F_1 could *not* just wait for the reply: the former leader *L* might have failed at point π, i.e., after having synchronized the extended state interval S_1, but before having started S_2 by sending the RPC request $\texttt{req}_B$. Instead, the new leader F_1 has to (re–)do any observable event unless it can deduce that the event already had been done by the former leader (as is the case if it *did* receive $\texttt{rep}_B$). This includes asynchronous RPCs, to which there are no replies. The same reasoning also applies to RPC answer messages sent by the leader. If F_1 fails after point ρ, F_2 will become the new leader. It cannot know whether or not F_1 already sent back $\texttt{rep}_A$ to the client, so F_2 has no choice but execute the reply (again).

The above implies that partitions must be able to deal with *duplicate messages* for both incoming RPC requests and incoming RPC answers. Messages must therefore be tagged with unique, system–wide identifier. Repeated answer messages are simply ignored. If a request message is delivered several times, the request is only executed the first time. Repeated requests then return the result of the first invocation, which implies that results must be retained. Retained results must be eventually garbage collected to avoid memory leaks, which can be done by piggybacking with each message information about which RPC answers the sender already got from the receiver of the message. The receiver of the message can then discard these retained answers.

A follower encountering an observable event (i.e., at the very beginning of an extended state interval) during replay does not have to redo the action because it *knows* with absolute certainty that the leader did indeed perform the operation. Duplicate sending of messages can only occur if the leader fails after an observable event, but before the next synchronization.

6 RAPIDS: Replicated Ada Partitions In Distributed Systems

RAPIDS [Wol98] is an implementation of the semi–active replication scheme based on piecewise determinism (described in sections 5 and 5.2) for the GNAT compiler. It is implemented within the run–time support as part of the PCS and is therefore largely transparent to the application.

Rapids consists of a core implementing the event log, offering an interface for logging and replaying events and for transferring the event log from the leader to the followers. Actually, there are three interfaces:

- an internal one for use by Garlic [KPT95], GNAT's PCS, for logging and replaying external events, and
- an interface through callbacks which is used by GNAT's tasking support GNARL for logging and replaying internal events, and finally
- an interface in a child package of `System.RPC` for use by the standard libraries, which also may have to log and replay events (e.g., `Ada.Calendar.Clock` or file accesses).

Garlic has been modified to include unique message identifiers in all its messages and to do duplicate message detection. The necessary group communication has been added as a new protocol to Garlic. This protocol currently is only an interface to a third–party view–synchronous group communication system called Phoenix [MFSW95]. The tasking support GNARL has also been modified by adding callbacks to RAPIDS for logging and replaying of internal events. However, not all tasking is synchronized: RAPIDS only logs events for application tasks, but not for tasks used within the PCS itself. (The PCS must do different things on a leader or a follower!)

7 Conclusion

We have presented a method to transparently replicate Ada 95 partitions to achieve fault tolerance. The approach is based on a piecewise deterministic computation model and employs semi–active replication. It can offer k–resiliency for arbitrary partitions in heterogeneous distributed systems.

The implementation of this approach, called 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 types are not yet handled.

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.

[Bir93] Birman, K. P.: "The Process Group Approach to Reliable Distributed Computing", *Communications of the ACM* ***36****(12)*, Dec. 1993, pp. 37 – 53.

[BN84] Birrell, A. D.; Nelson, B. J.: "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems* ***2****(1)*, 1984, pp. 39 – 59.

[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.

[HW90] Herlihy, M. P.; Wing, J. M.: "Linearizability: A Correctness Criterion for Concurrent Objects", *ACM Transactions on Programming Languages and Systems* ***12****(3)*, July 1990, pp. 463 – 492.

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

[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.

[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.

[Mos81] Moss, J. E. B.: *Nested Transactions: An Approach to Reliable Distributed Computing*, PhD Thesis MIT/LCS/TR–260, MIT, Cambridge MA, USA, Apr. 1981.

[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.

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

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

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

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