# Replication
# of
# Non–Deterministic Objects

THÈSE N$^{O}$ 1903 (1998)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Thomas Wolf

Ingénieur informaticien diplomé de l'École Polytechnique Fédérale de Zürich

originaire de Thayngen (SH)

acceptée sur proposition du jury:

Prof. Alfred Strohmeier, directeur de thèse

Prof. Hans–Heinrich Nägeli, rapporteur

Prof. André Schiper, rapporteur

Prof. Andy Wellings, rapporteur

Lausanne, EPFL

Novembre 1998

# Abstract

This thesis discusses replication of non–deterministic objects in distributed systems to achieve fault tolerance against crash failures. The objects replicated are the virtual nodes of a distributed application. Replication is viewed as an issue that is to be dealt with only during the configuration of a distributed application and that should not affect the development of the application. Hence, replication of virtual nodes should be *transparent* to the application.

Like all measures to achieve fault tolerance, replication introduces redundancy in the system. Not surprisingly, the main difficulty is guaranteeing the consistency of all replicas such that they behave in the same way as if the object was not replicated (*replication transparency*). This is further complicated if active objects (like virtual nodes) are replicated, and these objects themselves can be clients of still further objects in the distributed application.

The problems of replication of active non–deterministic objects are analyzed in the context of distributed Ada 95 applications. The ISO standard for Ada 95 defines a model for distributed execution based on remote procedure calls (RPC). Virtual nodes in Ada 95 use this as their sole communication paradigm, but they may contain tasks to execute activities concurrently, thus making the execution potentially non–deterministic due to implicit timing dependencies. Such non–determinism cannot be avoided by choosing deterministic tasking policies.

I present two different approaches to maintain replica consistency despite this non–determinism. In a first approach, I consider the run–time support of Ada 95 as a black box (except for the part handling remote communications). This corresponds to a non–deterministic computation model. I show that replication of non–deterministic virtual nodes requires that remote procedure calls are implemented as nested transactions. Unfortunately, effects of failures are not local to the replicas of a virtual node: when a failure occurs, nested remote calls made to other virtual nodes must be undone. Also, using transactional semantics for RPCs necessitates a compromise regarding transparency: the application must identify global state for it cannot be determined reliably in an automatic way. Further study reveals that this approach cannot be implemented in a transparent way at all because the consistency criterion of Ada 95 (linearizability) is much weaker than that of transactions (serializability). An execution of remote procedure calls as transactions may thus lead to incompatibilities with the semantics of the programming language. If remotely called subprograms on a replicated virtual node perform partial operations, i.e., entry calls on global protected objects, deadlocks that cannot be broken can occur in certain cases. Such deadlocks do not occur when the virtual node is not replicated. The transactional semantics of RPCs must therefore be exposed to the application.

A second approach is based on a piecewise deterministic computation model, i.e., the execution of a virtual node is seen as a sequence of deterministic state intervals. Whenever a non–deterministic event occurs, a new state interval is started. I study replica organization under this computation model (semi–active replication). In this model, all non–deterministic decisions are made on one distinguished replica (the leader), while all other replicas (the followers) are forced to follow the same sequence of non–deterministic events. I show that it suffices to synchronize the followers with the leader upon each *observable* event, i.e., when the leader sends a message to some other virtual node. It is not necessary to synchronize upon each and every non–deterministic event — which would incur a prohibitively high overhead. Non–deterministic events occurring on the leader between observable events are logged and sent to the followers just before the leader executes an observable event. Consequently, it is guaranteed that the followers will reach the same state as the leader, and thus the effects of failures remain mostly local to the replicas.

A prototype implementation called RAPIDS (Replicated Ada Partitions In Distributed Systems) serves as a proof of concept for this second approach, demonstrating its feasibility. RAPIDS is an Ada 95 implementation of a replication manager for semi–active replication for the GNAT development system for Ada 95. It is entirely contained within the run–time support and hence largely transparent for the application.

# Résumé

Cette thèse traite de la duplication d'objets non–déterministes dans des systèmes répartis afin de les rendre tolérants aux pannes, plus particulièrement aux défaillances par arrêt. Les objets dupliqués sont les nœuds virtuels d'une application répartie. La duplication de nœuds virtuels est censée être *transparente* pour l'application: on considère la duplication comme un mécanisme qui n'intervient qu'au moment de la configuration de l'application et qui ne devrait pas affecter le développement de celle–ci.

Comme toutes les mesures ayant comme but la tolérance aux pannes, la duplication introduit de la redondance dans le système. La difficulté principale à résoudre est évidemment de garantir la cohérence de tous les duplicata pour que l'ensemble de duplicata se comporte envers les autres objets comme un objet singulier (*transparence de duplication*). Le problème est encore amplifié si l'on considère la duplication d'objets actifs, qui peuvent eux–mêmes être des clients d'autres objets dans l'application répartie.

Dans cette thèse, les problèmes de la duplication d'objets actifs non–deterministes sont étudiés dans le cadre d'applications réparties mises en œuvre avec le langage de programmation Ada 95. Le standard ISO de ce langage définit un modèle pour l'exécution répartie fondé sur des appels à distance (RPC). Les nœuds virtuels en Ada 95 ne communiquent que par des appels à distance; ils peuvent néanmoins contenir des tâches, ce qui rend leur comportement non–déterministe en raison de certaines dépendances temporelles implicites. Ce genre de non–déterminisme ne peut pas être résolu en choisissant des règles d'ordonnancement de tâches déterministes.

Cette thèse présente deux approches différentes pour maintenir la cohérence des duplicata malgré ce non–déterminisme. Dans la première, le support d'exécution de Ada 95 est considéré comme une boîte noire, à l'exception de sa partie responsable du traitement de la communication entre les nœuds virtuels. Ceci correspond à un modèle de calcul non–déterministe. Je montre que la duplication de nœuds virtuels non–déterministes nécessite une sémantique de transactions imbriquées pour les appels à distance. Les effets de défaillances ne sont pourtant pas locaux à l'ensemble des duplicata d'un nœud virtuel: lors d'une défaillance, les appels à distance imbriqués faits à d'autres nœuds doivent être annulés. La sémantique transactionnelle nécessite également un compromis en ce qui concerne la transparence: l'application doit identifier son état global, car il ne peut pas être identifié avec certitude de manière transparente dans le support d'exécution. Mais le problème le plus sévère est sans doute qu'il est même impossible de mettre en œuvre cette approche d'une manière transparente car le critère de cohérence de Ada 95 (linéarisabilité) est beaucoup moins fort que celui de transactions (sérialisabilité). L'exécution d'appels à distance comme des transactions peut donc conduire à des incompatibilités avec la sémantique du langage de programmation lui-même. Si un sous–programme appelé à distance sur un nœud virtuel dupliqué accomplit des opérations partiels, c.-à-d. des appels d'entrées d'objets protégés globaux, des interblocages qui ne peuvent pas être résolus sont possibles. De tels interblocages ne surviendraient pas si le nœud virtuel n'était pas dupliqué! Il est donc indispensable que la nature transactionnelle des appels à distance soit révélée à l'application.

La deuxième approche se fonde sur un modèle de calcul à étapes déterministes *(piecewise deterministic computation model)* où l'exécution d'un nœud virtuel est modélisée par une séquence d'intervalles déterministes d'états. À chaque événement non–déterministe, un nouvel intervalle d'état commence. J'étudie l'organisation de duplicata dans ce modèle de calcul par le moyen de la duplication semi–active. Dans cette forme d'organisation, toutes les décisions non–déterministes se font sur un duplicata distingué (appelé *leader* ou meneur[1]) et tous les autres duplicata (les *followers,* ou suiveurs) sont forcés de suivre la même séquence d'événements non–déterministes. Je démontre qu'il suffit de synchroniser les suiveurs avec le meneur chaque fois que ce dernier est sur le point d'exécuter un événement *observable*, c'est–à–dire quand il envoie un message à un autre nœud virtuel. Il n'est pas nécessaire de les synchroniser à chaque événement — ceci entraînerait une baisse de performance prohibitive. Des événements non–déterministes qui surviennent entre deux événements observables sur le meneur peuvent être enregistrés dans un journal qui est envoyé aux suiveurs juste avant que le meneur exécute un événement observable. On garantit ainsi que les suiveurs vont atteindre le même état que le meneur. Avec ce procédé, les effets de défaillances restent presque complètement locaux aux duplicata.

Une première mise en œuvre, appelée RAPIDS (Replicated Ada Partitions In Distributed Systems), montre la faisabilité de la deuxième approche fondée sur le modèle de calcul à étapes déterministes. RAPIDS est un gestionnaire de duplicata pour la duplication semi–active pour le système de développement Ada 95 GNAT. Il est entièrement encapsulé dans le support d'exécution et il est donc largement transparent pour l'application.

1. La terminologie française est loin d'être normalisée. On trouve presque toutes les combinaisons possibles de termes dans la littérature: "copie primaire/secondaire", "leader/suiveur", "meneur/suiveur", "leader/follower", ...

# Zusammenfassung

Diese Dissertation analysiert die Replikation von nichtdeterministischen Objekten in verteilten Systemen zum Zwecke der Fehlertoleranz bezüglich Abstürzen (*crash failures*). Die replizierten Objekte sind hier die logischen Knoten (*virtual nodes*) einer verteilten Anwendung. Replikation von logischen Knoten soll für die Anwendung *transparent* sein: da ihre Funktionalität unverändert bleibt, wenn Knoten repliziert werden, soll auch ihre Entwicklung nicht tangiert werden. Erst bei der Konfiguration der verteilten Anwendung muss die Replikation von Knoten berücksichtigt werden.

Das Replizieren von Knoten führt — wie alle Methoden der Fehlertoleranz — zu Redundanz im System. Erwartungsgemäss ist die Koordination der Repliken somit das grösste zu lösende Problem. Ziel dieser Koordination ist *Replikationstransparenz*, d.h. ein repliziertes Objekt soll sich gegenüber dem Rest des Systems so verhalten, als ob es nicht repliziert wäre. Betrachtet man die Replikation von aktiven Objekten, so wird dies noch erschwert, wenn ein repliziertes Objekt selbst Dienste von weiteren Objekten benutzt.

Die Problematik der Replikation von aktiven, nichtdeterministischen Objekten wird in dieser Dissertation anhand der Programmiersprache Ada 95 untersucht. Der ISO–Standard dieser Sprache definiert ein Modell für die Programmierung und Ausführung von verteilten Anwendungen, das im wesentlichen auf dem Konzept des Remote Procedure Calls (RPC) beruht. Die einzelnen Knoten einer verteilten Anwendung in Ada 95 kommunizieren zwar ausschliesslich durch RPC, können jedoch ohne weiteres Tasks beinhalten. Aufgrund impliziter Zeitabhängigkeiten wird das Verhalten eines solchen Knotens nichtdeterministisch, selbst wenn ein deterministischer Scheduling–Algorithmus für Tasks gewählt wird.

In dieser Dissertation stelle ich zwei verschiedene Methoden für die Koordination nichtdeterministischer Repliken vor. In einem ersten Ansatz wird die Laufzeitbibliothek von Ada 95, mit Ausnahme der Unterstützung der Kommunikation in verteilten Anwendungen, als "black box" betrachtet. Dies entspricht einem nichtdeterministischen Ablaufmodell. Das Replizieren nichtdeterministischer Knoten führt dazu, dass Remote Procedure Calls die Semantik von geschachtelten Transaktionen haben müssen. Dabei sind jedoch die Auswirkungen eines Absturzes nicht auf die Gruppe der Repliken eines Knotens beschränkt: tritt ein Absturz während der Ausführung eines Remote Procedure Calls auf, so müssen alle geschachtelten RPCs zu anderen Knoten zurückgesetzt werden. Der gravierendste Mangel dieses Ansatzes ist jedoch, dass er nicht in transparenter Art und Weise implementiert werden kann. Einerseits muss eine Anwendung ihre globalen Daten identifizieren, da diese nicht zuverlässig automatisch erkannt werden können, andererseits ist die Konsistenzbedingung von Ada 95 (Linearisierbarkeit) weniger strikt ist als jene von Transaktionen (Serialisierbarkeit). Damit kann eine Ausführung von Remote Procedure Calls als Transaktionen zu Inkompatibilitäten mit der Semantik der Programmiersprache führen, falls in einem replizierten Knoten partielle Operationen, d.h. Aufrufe von Entries von globalen Protected

Objects, ausgeführt werden. In einem solchen Fall können Deadlocks, die nicht aufgelöst werden können und zu denen es nicht gekommen wäre, wenn der Knoten nicht repliziert worden wäre, auftreten! Somit muss die transaktionelle Implementierung von Remote Procedure Calls zwangsläufig der Anwendung sichtbar gemacht werden, damit diese den neuen Rahmenbedingungen Rechnung tragen kann.

Der zweite Ansatz geht von einem stückweise deterministischen Ablaufmodell *(piecewise deterministic computation model)* aus. Dabei wird die Ausführung einer Anwendung (bzw. eines Knotens) durch eine Sequenz von deterministischen Zustandsintervallen abgebildet. Mit jedem nichtdeterministischen Ereignis wird ein neues Zustandsintervall begonnen. Die Koordination mittels semi–aktiver Replikation wird untersucht. Dabei werden alle nichtdeterministischen Entscheidungen auf einer ausgezeichneten Replik (dem *Leader*) getroffen, während alle anderen Repliken (die *Followers*) diesen Entscheidungen zu folgen haben, d.h. alle nichtdeterministische Ereignisse in der selben Reihenfolge ausführen müssen. Es wird gezeigt, dass eine Synchronisation zwischen dem Leader und seinen Followers bei jedem *extern sichtbaren* Ereignis, d.h. wenn der Leader eine Nachricht an einen anderen Knoten sendet, ausreichend ist. Es ist nicht erforderlich, die Repliken bei jedem nichtdeterministischen Ereignis zu synchronisieren — was auf Grund des hohen Aufwands kaum praktikabel wäre. Das Auftreten nichtdeterministischer Ereignisse auf dem Leader zwischen zwei extern sichtbaren Ereignissen wird in einem Log aufgezeichnet. Bevor der Leader ein extern sichtbares Ereignis ausführt, wird dieses Log allen Followers übermittelt. Damit kann sichergestellt werden, dass diese den gleichen Zustand wie der Leader erreichen werden. Auf diese Art und Weise können die Auswirkungen von Abstürzen grösstenteils in der Gruppe von Repliken verkapselt werden.

Die Implementation des Prototyps RAPIDS (Replicated Ada Partitions In Distributed Systems) zeigt, dass dieser zweite Ansatz des stückweisen Determinismus machbar ist. RAPIDS ist ein Replikationsmanager für semi–aktive Replikation für GNAT (ein Entwicklungssystem für Ada 95), der gänzlich in den Laufzeitbibliotheken verkapselt ist. Die Replikation von Knoten ist somit weitgehend transparent für Anwendungen.

# Acknowledgements

I could not have done this work without the help of others. I am thankful for the opportunity to express my gratitude to them here.

I thank Professor Alfred Strohmeier for having accepted me as a PhD student and for supervising this research work. He has given me a lot of freedom in approaching the topic of this thesis, and has given me ample opportunities to meet some of the experts in the field by inviting them to our lab or by sending me to renowned conferences. I have profited immensely from these encounters. I also thank Professor Strohmeier for having read draft versions of this thesis; I think it has benefited greatly from his criticisms.

I am grateful to the jury members for having accepted to serve on my examination board and for the time they invested to read and evaluate this work.

I am indebted to Professor Andy Wellings, who had asked the right questions at the right time when he was visiting the Software Engineering Lab in summer 1997. I also thank him for inviting me to participate in the $8^{th}$ International Real–Time Ada Workshop (IRTAW–8). The interactions with him proved very fruitful indeed and brought this work back on the right track.

I also thank all the members, past and present, of the Software Engineering Lab at EPFL for the pleasant atmosphere. Special thanks go to Stéphane Barbey, who helped me find my ways in Ada 95, and to my office mate Jörg Kienzle for the many discussions, his careful review of this thesis, and for his friendship in general. I also thank Nicolas Guelfi for relieving me from some of my responsibilities in the preparation of the software engineering course 1998/99, for proofreading a first draft of the French abstract, and generally for creating a friendly and convivial ambiance.

Very special thanks go to Hannelore — without her constant encouragement and support I might not have accomplished this work. And I thank Fabian for always smiling at me, even when I was at times a bit short–tempered and preoccupied with my work in the final phase of this thesis. Anyway, I thank them both for bearing so well my occasional grumpiness when stress got the better of me.

This work has been supported in part by a grant of the Board of the Swiss Federal Institute of Technology.

*To Hannelore*

*for her constant support,*

*and to Fabian*

*for his smiles.*

# Table of Contents

# List of Figures

## Part I: Motivation and Fundamental Concepts

### Chapter 2: Fault Tolerance in Distributed Systems



### Chapter 3: Distributed Systems in Ada 95



## Part II: Replication in Ada 95

### Chapter 4: Prelude



### Chapter 5: Non–Deterministic Replicas

# Part I

# Motivation and Fundamental Concepts

# Chapter 1:
# Introduction

## 1.1 Context and Objectives

Fault tolerance has the goal of allowing an application to continue to work — maybe in a degraded fashion — even when failures in the system executing it occur. This is especially important in distributed systems, where a distributed application might be taken down entirely by the failure of even only one node executing a part of it. Leslie Lamport's famous aphorism testifies to this:

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*
>
> *— Leslie Lamport, 1987 (attributed)*

This expresses succintly the behavior fault tolerance strives to avoid.

This thesis investigates the replication of virtual nodes in distributed applications as a means of achieving fault tolerance in the presence of crash failures. Replication is an often–studied topic, but most previous work was done in the context of either data replication (e.g. caching, with the primary goal of improving performance) or the replication of deterministic objects, e.g. the state machine approach of Schneider [Sch90].

Modern software systems and their programming languages often do not a priori fulfill the stringent condition of determinism. Applications can be multi–threaded, and server nodes in a distributed application can handle several requests concurrently. Using the state machine approach in such a case often amounts to crippling the system by effectively disregarding potential gains that multi–threading might bring and handling requests one after another. This not only hurts the throughput of the system but also artificially diminishes the expressive power of modern–day programming languages as the possible interactions between requests to a node are restricted.

I have chosen to examine the problems of replication in distributed systems using the programming language Ada 95 [ISO95]. Ada is mostly known for its clean integration of concepts and structures for expressing concurrent activities in the programming language in the form of tasks. But with the revised ISO standard for Ada, it also precisely specifies a model for distributed execution: this is a solid foundation to build distributed applications upon. Yet this language standard itself makes only a token reference to the subject of fault tolerance in distributed systems:

> *"An implementation may allow separate copies of an active 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)]*

This despite the fact that one of the strongholds of Ada is the domain of dependable systems. The programming language has a long history and excellent reputation in the development of fault–tolerant and safety–critical systems, hence the interest of studying this point in detail.

The above quote from the ISO language standard [ISO95, E.1(12)] clearly implies that replication of virtual nodes (called "partitions" in Ada 95) can be provided in a manner *transparent* to the application. Configuration of a distributed application is seen in the language standard as an activity that takes places *after* the application has been programmed and compiled. By giving permission to replicate some virtual nodes *at configuration time*, the standard takes the view that replication is a non–functional requirement that can be dealt with only at the very end of the development process of a distributed application. This implies *replica transparency*: the application level of the replicated partition itself should not be aware of replication. The quoted paragraph also addresses *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. In this thesis, I examine this point in detail and try to answer the question to what extent the assumption of transparent a posteriori replication holds.

Also, the powerful semantics of Ada 95 is usually implemented in a run–time support that is interposed between the application and the underlying operating system. The semantics of threads as they are provided by modern operating systems for instance does not match the requirements of tasking as it is defined for Ada 95. The run–time support maps in this case the semantic construct of a "task" onto the much lower–level entity "thread" and at the same time implements the added functionality of tasks using whatever primitives the operating system offers. The presence of such a run–time support offers the unique possibility of implementing replication for Ada 95 not only in a *transparent* way, but also in a largely *portable* way, whereas — had I chosen another programming language — an implementation would have in all likelihood depended much more on the peculiarities of a particular operating system.

## 1.2 Contributions of this Thesis

The main contributions of this work are the following:

- An analysis of replication for fault tolerance of objects that may behave non–deterministically. This analysis is done in the context of Ada 95, where the objects correspond to partitions, i.e., virtual nodes.

  I present two different approaches to replication of non–deterministic objects. A first approach is based upon the transparent implementation of remote procedure calls as nested subtransactions. I show that this method is not suitable for achieving transparent replication because of an incompatibility of the semantics of concurrent executions as defined in Ada 95 and the serializability model of transactions.

  A second approach assumes a piecewise deterministic computation model, in which deterministic execution state intervals are separated by non–deterministically occurring events. I show that this model preserves the correctness of Ada 95 partitions and can be used to offer replication in a transparent way.

- A prototype implementation of a replication manager for Ada 95 partitions called RAPIDS (Replicated Ada Partitions In Distributed Systems).

  This implementation is a realization of the second approach analyzed, i.e. based on a piecewise deterministic model of computation. RAPIDS implements a semi–active replication scheme and employs event logging and replay for maintaining replica consistency by making all replicas go through the same execution history. Acceptable performance is achieved by basing replica synchronization on the notion of observable events: as long as only events local to the partition occur, synchronization is not necessary.

## 1.3 Thesis Organization

This doctoral thesis deals with advanced topics, and I assume readers have a solid background in fault–tolerant computing. Nevertheless I have tried to structure the thesis in such a way that it be comprehensible for any reader with a general background in computer science. In particular, I'll briefly recall basic principles before building upon them.

The structure of this thesis closely reflects the progress of my work. It is organized in two main parts. Part I lays the basic foundations necessary for the apprehension of the later discussion of replication in part II.

Chapter 2 is a concise overview of the basic concepts of fault–tolerant computing with an emphasis of concepts for fault tolerance in distributed systems. After introducing the basic terminology (following [Lap85]), I present the foundations for replication in distributed systems: consistency models, group communication, and replication management. The chapter closes with an overview of the main software fault tolerance techniques.

Although Ada exists for more than 15 years now, the recent revision of the language standard enhanced the language considerably. In chapter 3, I give a brief review of the most important new features in Ada 95: support for object–oriented programming and novelties in the tasking system. I also give an overview of the model of distributed applications defined in the Ada 95 language standard.

Part II of this thesis starts in chapter 4 with a description of the objectives of this thesis and a presentation of the system model I assume. I discuss various choices of units of replication, and I show that a distributed Ada 95 application is inherently non–deterministic due to implicit timing dependencies.

Chapter 5 describes a first approach to replication in Ada 95. It examines the problem under the assumption of a non–deterministic computation model. I show that remote calls must have transactional semantics in this case. Closer analysis reveals that transactional serializability does not integrate well with the model of tasking defined in Ada 95. I show that transparently transforming RPCs into nested transactions when a partition is replicated may lead to deadlocks that would not occur if the partition were not replicated and RPCs thus did not have transactional semantics. Furthermore, these new deadlocks cannot be broken. The conclusion is that transactions cannot be added in a transparent way to Ada 95, but should rather be offered on the language level.

In chapter 6, I have therefore adopted a different computation model based on the assumption of piecewise deterministic executions. I show that this model also is a valid description of executions in Ada 95, and that the semantics of the programming language are such that replicas can be synchronized by event logging only — implicit timing dependencies do not influence replica consistency. I define the set of events and show that synchronization of replicas is only needed prior to events that are observable by other partitions.

Chapter 7 gives the state of the art in this thesis' domain in the form of a brief overview of some relevant previous systems, which can be broadly classed as either transactional or based on a piecewise deterministic model.

Chapter 8 is a detailed description of RAPIDS (Replicated Ada Partitions In Distributed Systems), an implementation of a replication manager for the GNAT development system based on the piecewise deterministic computation model described in chapter 6. It begins with a short description of the structure of the run–time support and then goes on to show which parts are affected by replication management. It also describes in detail how event logging and replica synchronization work.

Finally, chapter 9 gives a conclusion, summarizing the main results of this work and indicating some areas for future work.

# Chapter 2:
# Fault Tolerance in Distributed Systems

In this section, I give a brief introduction to fault–tolerant computing in general, before discussing the use of replication for fault tolerance.

## 2.1 Fault Tolerance

What exactly is meant by fault tolerance always depends upon the context in which one operates. It is therefore important to first define this context and the domain–specific terms used.

### 2.1.1 Terminology

To discuss fault tolerance meaningfully, a definition of correct behavior of a program is needed — otherwise, how could one know that something went awry? For the purposes of fault–tolerant computing, the *specification* of the program is considered this definition of correct behavior: as long as the program meets its specification, it is considered correct. A deviation from the specification is considered a *failure*. A failure is therefore the observation of an erroneous system state, i.e., a failure is caused by an error. An *error* is that part of the system state that leads to a failure of the system. An error itself is caused by some defect in the system; those defects that cause observable errors are called *faults*. There may be defects in the system that remain undetected; only those that manifest themselves as errors are considered faults. Likewise, an error not necessarily leads to a failure: it may be a *latent* error [Lap85]. Only when the error in the system state causes the system to behave in a way contradictory to its specification, a failure occurs.

The goal of fault tolerance is to avoid system failure in the presence of faults. When an error occurs, it must be corrected to avoid a later potential failure: corrective actions have to be taken to restore the system state correctly.

### 2.1.2 Fault Classification

Faults can be characterized in various ways. One considers the temporal characteristics of a fault. A *transient* fault has a limited duration, e.g. a temporary malfunction of the system, or a fault due to external interference. If a transient fault occurs repeatedly, it is called an *intermittent* fault. In contrast, *permanent* faults persist, i.e. the faulty component of the system will not work correctly again unless it is replaced.

Another way to classify faults is to consider the software lifecycle phase in which they occur. Here, one can distinguish *design faults* (in particular software design faults) from *operational faults* occurring during the use of the system.

### 2.1.3 Failure Semantics

Failures, i.e. deviations from a program's specification, can manifest themselves in various ways [Cri91]:

- *Timing* failures can occur in real–time systems if the system fails to respond within the specified time slice. Both early and late responses are considered timing failures; late timing failures are sometimes called *performance* failures.
- *Omission* failures occur when the system doesn't respond to a request when it is expected to do so.
- A *crash* failure occurs when the system stops responding at all. One generally distinguishes *fail–silent* and *fail–stop* behavior: with the latter, the clients of the system have a means to detect that it has failed.
- If a failed system can behave arbitrarily, it is said to exhibit *byzantine* failure semantics [LSP82].

Byzantine failures are the most general failure type. [LSP82] studies the problem of byzantine agreement in synchronous (see 2.2.1) distributed systems, where nodes that communicate by message passing over a fully connected network and that may be subject to byzantine failures must agree within bounded time on a common decision. It is shown that this is possible only if less than one third of the nodes fail. The problem is complicated because a failed node might maliciosly masquerade as another one and send confusing messages on its behalf.

If messages are authenticated, the sender of a message can always be determined reliably, and any tampering with messages or masquerading a failed node might attempt can be detected. Cryptographic techniques such as digital signatures can be used to achieve this. In this case, agreement can be reached for an arbitrary number of failures [LSP82, BMD93].

These failure semantics form a hierarchy: byzantine failures are the most general model, and subsume all others as shown in fig. 2.1 below.

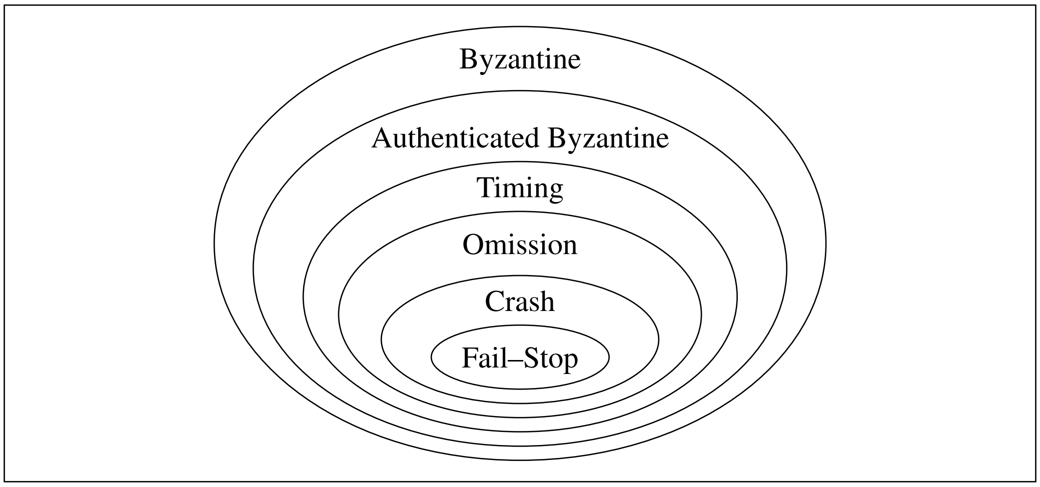


**Fig. 2.1:** Failure Semantics Hierarchy

### 2.1.4 Error Processing

As mentioned above, errors detected in the system state must be corrected to avoid a potential system failure later on. Of course, the fault(s) causing the error also should be treated, which means that the reason for the error must be identified and then the defect be corrected in order to avoid that the fault causes more errors. Fault diagnosis and removal is quite different from error processing and is beyond the scope of this thesis.

Once an error is detected, there are several techniques that can be used to treat it. I distinguish preventive (error *compensation*) and corrective (error *recovery*) measures.

*Error masking* is the main preventive fault tolerance technique. It exploits redundancy to detect errors and to mask them; a common example is *triple modular redundancy* (TMR): a fault–tolerant component consists of three replicas, the output of the component is the result of some comparator function of the three replicas' individual outputs. Voting (i.e., taking the majority of replies) is one possible comparator function, but depending on the context and the failure semantics of the replicated component, other functions such as taking the average might be adequate.

Corrective methods try to bring the system into a correct state again once an error has been detected. There are two base cases:

- *Forward error recovery* attempts to construct a coherent, error–free system state by applying corrective actions to the current, erroneous state.
- *Backward error recovery* replaces the erroneous system state with some previous, correct state.

Backward error recovery requires that a previous correct state exists: such systems periodically keep a copy of a coherent state (*recovery point* or *–line*), to which they can *roll back* in case of an error. Backward error recovery is a general method: because it re–installs a previous, hopefully correct system state, it does not depend on the nature of the error nor on the application's semantics. Its main drawback is that it incurs a certain overhead even in failure–free execution because recovery points have to be established from time to time.

Forward error recovery requires that a more or less accurate damage assessment be made. The error must be identified in order to apply corrective actions to exorcise it. This diagnosis for forward error recovery depends on the particular system.

### 2.1.5 Error Confinement

(Software) systems are not monolithic; they usually consist of several components or subsystems, and fault tolerance approaches must account for that. Different approaches may be applied to different components. The composite nature of systems also means that the classification of fault, error, and failure is not absolute: a given component may perceive the failure of a sub–component as a fault and have its own fault tolerance techniques in place to handle it.

This hierarchic model of a system gives rise to the notion of error confinement: the system is structured in regions beyond which the effects of a fault should not propagate undetected. This implies that a given component be accessible to other components only through a well–defined (and preferably narrow [Kop97]) interface. Different error confinement regions may employ different means to achieve fault tolerance depending upon the failure semantics the system component should adhere to according to its specification as well as on the failure semantics of its sub–components.

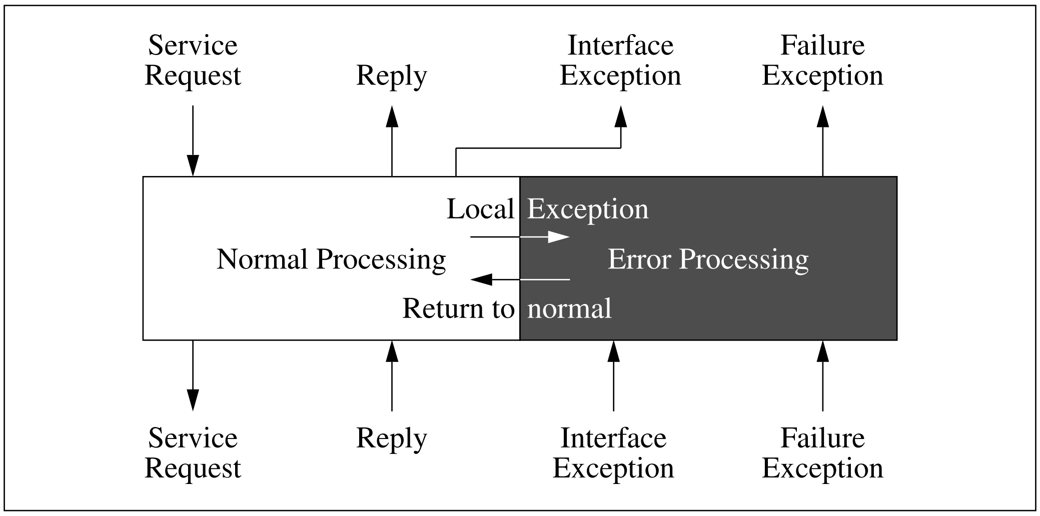


**Fig. 2.2:** Idealized System Component

The idealized error–confining system component [RX95] is shown in fig. 2.2. The component offers services that may return replies to the component that made a service request. If a request is malformed, the component signals this by raising an interface exception, otherwise it executes the request and produces a reply. If an exception signaling an error occurs, error processing is activated in an attempt to handle the error. If it can be dealt with, normal processing in the component resumes; if not, the component itself signals its failure by an exception. It is immaterial whether exceptions are true exceptions in the sense of Ada 95 or are indicated using exceptional replies to requests. It is even possible that some entity external to the system component observes its failure and initiates appropriate error processing in the users of the component.

## 2.2 Distributed Systems

### 2.2.1 Characteristics of Distributed Systems

A distributed system is a set of processing nodes (*physical nodes*, i.e., computers) that are interconnected by a network allowing them to communicate with each other. I assume that physical nodes do not share memory: they only communicate by sending messages over the network.

Besides this physical model, a distributed system can be modeled by its temporal characteristics. If no timing assumptions at all are made, the system is dubbed *asynchronous*. In a *synchronous* distributed system, the timing behavior is constrained by the following assumptions [HT94]:

- there is a known finite upper bound for the message transmission delay, and
- there is a known finite upper bound for the time required by any node to execute a step, and
- on every node, there is a local clock with a known finite upper bound for clock drift with respect to real time.

It is therefore possible to use time-outs to detect failures reliably in synchronous systems[1]; in asynchronous systems, only *failure suspicions* can be made because a slow node or communication link cannot be distinguished from a failed one.

### 2.2.2 Distributed Applications

I define a distributed application as an application executing on a distributed system. A distributed application is composed of *virtual nodes*; each virtual node is allocated to some

1. Note that all three conditions must be met to be able to detect failures reliably using time-outs. If any of them is violated, a time-out not necessarily signifies a failure; it might be that the transmission delay was longer than expected, or that the receiving node was slow, or that the local clock ran too fast.

physical node, where it is executed. The virtual nodes of an application communicate with each other over the network by sending *messages* through *channels* to accomplish the application's task. A channel is a logical communication link between two virtual nodes; for the purpose of this thesis, channels are assumed to be *reliable*. 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 through retransmissions of messages.

Distributed applications may exhibit a wide range of communication patterns between their virtual nodes depending on the way they are structured and the task they shall accomplish. I assume throughout this thesis a *client–server* model: each virtual node offers services that can be invoked by sending a message to the node. The node offering the service is the *server* node, the one requesting the service by sending the message is the *client* node. In order to fulfill the service request, the server node may send back a result to the client.

The classification of client and server is only meaningful in the context of one particular service invocation; it is by no means a global, static role. A virtual node acting as a server for one service may well need a service of yet another node to implement its own functionality: it plays the roles of client and server at the same time, but for two different service requests. Nevertheless, it is convenient to distinguish these cases. Following the terminology of Mazouni [Maz96],

- C–Components are virtual nodes that act as clients only,
- S–Components act as servers only, and
- CS–Components act as both.

## 2.3 Replication in Distributed Systems

Replication is one means to achieve fault tolerance in distributed systems by masking errors in the replicated component. In this section, I discuss the fundamental concepts needed in replication: replica consistency, group communication, and replica organization.

For the purposes of this section, a distributed application is viewed as a set of cooperating objects, where an *object* could be one of many things: a whole virtual node, a process, a variable, or an object in the sense of object–oriented programming. In order to render an object fault–tolerant, it is *replicated*: the application maintains several identical copies of the object that behave and are manipulated as if there existed only one copy. A copy of such a replicated object is called a *replica*. A failure of a replica can be masked thanks to the remaining replicas, which ensure that the object remains available despite the failure. Replication for fault tolerance should be transparent: clients of the object perceive the whole set of replicas as if only one single copy of the object existed. The *replicated object* is this ideal object that hides and encapsulates the individual replicas. Clients cannot distinguish replicated objects from solitary objects.

### 2.3.1 Replica Consistency

Whenever a system employs replication in some form, the issue of consistency between the replicas of an object comes up. This is not limited to replication for fault tolerance, e.g., the issue is also raised in distributed shared memory (DSM) implementations, where objects or memory units (pages) may be cached on several nodes to improve performance. Consistency is described by a *consistency model*, a specification on the behavior users of an object may observe. Many consistency models have been proposed in the literature; for a survey focussing on DSM, see [Mos93] or [Tan95].

Replication for fault tolerance requires a *strong* consistency criterion: sequential consistency or linearizability. Strong consistency gives users the illusion of a non–replicated object.

To reason about consistency criteria[1], the system is modeled as a set of concurrent sequential processes $\mathbf{P} = \{p_1, ..., p_n\}$ that communicate by accessing shared objects $\mathbf{O} = \{x_1, ..., x_m\}$, which are typed. An object can only be modified through the operations defined by its type. The behavior of an object $x$ is defined by its sequential specification, i.e., the sequences of operation executions on $x$ allowed by the semantics of the object's type in the absence of concurrency and failures; it can be seen as the set of all possible acceptable object histories of $x$. An operation on an object is *total* if it is defined for every state of the object, otherwise it is *partial*. Partial operations can be implemented by waiting until the object's state is such that the operation is defined.

The execution of the system is modeled by a *history* $\mathbf{H}$, which is a finite sequence of events (invocations $inv(o)$ and responses $res(o)$ of operations $o$ on objects). A history is *complete* if it contains for each invocation a matching response. A history is *sequential* if its first event is an invocation, and each invocation is immediately followed by a matching response. An *object subhistory* $\mathbf{H}|_x$ is the subsequence of operations on object $x$ in $\mathbf{H}$. A history $\mathbf{H}$ is *legal*, if for all $x$, $\mathbf{H}|_x$ belongs to the sequential specification of object $x$. A *process subhistory* $\mathbf{H}|_p$ is the subsequence of operations invoked (including the matching responses) by process $p$ in $\mathbf{H}$. Two histories $\mathbf{H}$ and $\mathbf{H}'$ are *equivalent* if for all $p \in \mathbf{P}$, $\mathbf{H}|_p = \mathbf{H}'|_p$. The order of events in $\mathbf{H}$ is denoted by the relation $<$. Note that $<$ does not enforce an order on $\mathbf{H}$; $e < f$ is just a notational convenience to express that event $e$ precedes event $f$ in $\mathbf{H}$. (As processes perform operations sequentially, $inv(o_1) < res(o_1) < inv(o_2)$ will of course always hold for two operations $o_1$ and $o_2$ performed in sequence by the same process, but events of operations related to different processes may appear in any order in a history.)

1. For some examples and for a more complete discussion, see [HW90].

The *sequential consistency* criterion was first proposed in [Lam79] and later formalized in various ways. The definition given here is due to [AW94]:

*Definition 2.1: Sequential Consistency*

- A history **H** is sequentially consistent if there exists an equivalent legal sequential history **S**.

Note that sequential consistency imposes no ordering on the events in **H** or **S** except the order given by the sequential execution of processes (i.e., by the process subhistories $\mathbf{H}|_p$). Operations invoked by different processes appear unordered, i.e., they may be in different orders in **H** and **S**.

*Linearizability* was introduced in [HW90] as an alternative consistency criterion. It adds an ordering constraint to the above definition by defining an ordering relation $<_L$ on operations:

- $o_1 <_L o_2$ if $res(o_1)$ precedes $inv(o_2)$ in **H**, i.e., if $res(o_1) < inv(o_2)$.

*Definition 2.2: Linearizability*

- A history **H** is *linearizable* if it can be completed to some history **H'** by appending missing response events so that
  - **H'** is equivalent to some legal sequential history **S**, and
  - $<_L \subseteq <_S$.

In other words, the equivalent sequential history **S** must preserve the relative "real–time" order of operations performed on objects, i.e., the order in which they appeared in **H**. I will use the notation $\mathbf{H}_{<_L}$ to denote a history **H** satisfying the order $<_L$.

Although linearizability is a stronger consistency criterion than sequential consistency (any linearizable history also is sequentially consistent), it turns out that linearizability is simpler to implement because it is a *local* property, i.e., a system's history **H** is linearizable if the histories $\mathbf{H}|_x$ of all its individual objects $x$ are linearizable. Linearizability is also a *non–blocking* property in the sense that the invocation of a total operation never causes the process making the invocation to block. For a proof see [HW90].

More recently, a consistency criterion based on the generalization to object–based systems of Lamport's "happened before" relation [Lam78] has been proposed, called *NRT–Linearizability*[1] in [Pac95] and *Normality* in [GR96]. (Both names denote exactly the same concept.) This is a weaker criterion than linearizability that still maintains the desirable properties of locality and non–blocking; yet it is stronger than causal consistency [ABHN91] in that it requires all processes to agree on the same history. (Causal consistency, as defined for shared memory systems, allows different processes to "see" different orders of not causally related write operations.)

1. "NRT" stands for "Non Real–Time".

### 2.3.2 Replica Determinism

An object is *deterministic* if its state changes only in function of the sequence of operations applied to the object and if each operation itself is reproducible, i.e., if it were invoked multiple times on the same state, the resulting new state would always be the same. The *state machine* [Sch90] model is based upon deterministic objects. A state machine encapsulates state that can be modified only through deterministic commands that execute atomically with respect to each other. When a client makes a request to a state machine, the latter handles the request by executing the corresponding command, which may generate some results or output.

A sufficient condition to ensure linearizability for deterministic replicated objects (state machines) is to make the replicas handle the same set of requests in the same order [Sch90, GS96], i.e.

- *Atomicity:* If one correct replica handles a request **r**, all correct replicas handle **r**.
- *Order:* If one correct replica handles a request **r** before a request **s**, all correct replicas handle **r** before **s**.

If the replicas initially are in the same state, the above conditions guarantee that all replicas will evolve identically and clients observe a consistent behavior of the replicated object.

### 2.3.3 Group Communication

Standard point–to–point communication often is not well adapted to the needs of distributed applications, especially when considering replication. The *group* concept is a communication abstraction offering more powerful services: a membership service, and a multicast facility. The membership service gives each member (consistent) information about the other members of the group, whereas the multicast facility offers a way to send messages to all group members at once.

A fundamental problem in group communication (or in distributed systems in general) is the *consensus* problem: given a set of processes[1], each process proposes a value $v_i$. They then have to decide on one common value **v** of the values $v_i$. Consensus can be defined by the following three properties:

*Definition 2.3: The Consensus Problem*

- *Agreement:* If a correct process decides on a value **v**, eventually all correct processes decide on **v**.
- *Integrity:* If a correct process decides on **v**, this value **v** has been previously proposed by some process ($\mathbf{v} \in \{v_1, ..., v_n\}$).

1. In order to be consistent with the terminology in the literature, I'll use the term "process" in this sub–section instead of "virtual node". A distributed system is viewed as a set of processes communicating by sending messages over reliable communication channels.

- *Termination:* Each correct process eventually decides exactly once.

Fischer, Lynch, and Paterson have shown in [FLP85] that consensus cannot be solved deterministically in a purely asynchronous system with even only one crash failure. This impossibility result has motivated Chandra and Toueg to introduce the notion of an asynchronous system augmented by an unreliable failure detector [CT91].

An *unreliable failure detector* can be seen as a distributed oracle giving each process in the system information about the correctness or failure of the other processes. This information cannot be precise in an asynchronous system; the failure detector can only *suspect* other processes to have failed. Failure suspicions are not static: a failure detector may cease to suspect a process to have failed if it discovers that its suspicion was wrong. Chandra and Toueg have defined in [CT95] different failure detectors that can be classified by two properties:

- *Completeness:* Eventually, every failed process is permanently suspected. If it is permanently suspected by *all* correct processes, the failure detector is said to satisfy *strong completeness*; if it is suspected by *some* correct process, the failure detector satisfies *weak completeness*.
- *Accuracy* defines the type of mistakes a failure detector may make in suspecting a process:
  – *Strong accuracy:* no process is suspected before it fails.
  – *Weak accuracy:* some correct process is never suspected.
  – *Eventual strong accuracy:* there is a time after which correct processes are not suspected by any correct process.
  – *Eventual weak accuracy:* there is a time after which some correct process is never suspected by any correct process.

These two criteria define eight possible failure detectors. Of particular interest are the failure detector classes $\Diamond S$ (called *eventually strong*, satisfying strong completeness and eventual weak accuracy) and $\Diamond W$ (called *eventually weak*, satisfying weak completeness and eventual weak accuracy). It is shown in [CHT94] that $\Diamond W$ is the weakest failure detector for solving consensus in an asynchronous system, and in [CT95] that a failure detector of class $\Diamond S$ can be constructed from one of class $\Diamond W$.

The *multicast* primitives offered by a group allow a message to be sent to the group as a whole, without explicitly having to know its individual members. The group serves also as a naming construct: each group has a name, and messages can be sent (multicast) to the group using this name as the recipient. There are various multicast primitives that can be defined.

The simplest fault–tolerant multicast primitive is the *reliable* multicast.

*Definition 2.4: Reliable Multicast*

A multicast is said to be reliable if it satisfies the following three properties:

- *Validity:* If a correct process multicasts a message *m*, it eventually delivers it.
- *Agreement:* If a correct process delivers a message *m*, all correct processes eventually deliver *m*.
- *Integrity:* Every correct process delivers a message *m* only once, and only if *m* was previously multicast.

Reliable multicast gives an all–or–nothing guarantee: either all correct processes deliver a message, or none do. However, it doesn't specify anything about the order in which processes deliver messages sent by reliable multicast. Most uses of groups need some ordering guarantees, though.

*FIFO* multicast extends reliable multicast with the additional condition that if a process multicasts a message $m_1$ before a message $m_2$, then no correct process delivers $m_2$ unless it has also delivered $m_1$[1]. Messages from different processes are unordered.

*Causal* multicast is similar. The additional condition in this case is based on causality as defined by Lamport's "happened before" relation [Lam78]: if a multicast of a message $m_1$ "happens before" the multicast of a message $m_2$, no correct process delivers $m_2$ without delivering $m_1$ first. Causal multicast implies FIFO multicast.

*Totally ordered* multicast (called *atomic* multicast in [HT94]) extends reliable multicast with a guarantee of a different quality: if a correct process delivers a message $m_1$ before it delivers a message $m_2$, then all correct processes deliver $m_1$ before $m_2$. Totally ordered multicast is equivalent to the consensus problem [CT91].

Note that total order is only concerned with the delivery of messages, whereas FIFO and causal multicasts also depend upon the relation between the sending (multicasting) of the messages. Total order does not impose any particular order on the delivery of messages except that all correct processes must choose the same order. This suggests that the FIFO or causality conditions can be combined with total order, yielding totally ordered FIFO multicast and totally ordered causal multicast.

*View–synchronous* group communication is a communication abstraction described first in the context of the Isis project [Bir85], where it is called *virtually synchronous* group communication. It is based on the notion of *views*: all correct processes in a group have a consistent view of the composition of the group. As processes join and leave the group (either voluntarily or because they fail or are suspected to have failed), the group composition changes over time, leading to a sequence of views $\mathbf{V} = \{\mathbf{V_0}, ..., \mathbf{V_i}, \mathbf{V_{i+1}}, ...\}$. The installation of a new view in the correct processes in the group, i.e., the delivery of a view, is called a *view change*.

1. See [HT94] for some comments on the subtleties of this formulation.

View–synchronous communication replaces the basic reliable multicast with a stronger primitive called *view–synchronous multicast*, which imposes a total order on the delivery of messages with respect to view changes. It is defined by the basic three conditions for reliable multicast, plus a fourth condition

*Definition 2.5: View Synchrony*

- *View synchrony:* If a process in view $\mathbf{V_i}$ delivers a message *m* and then delivers view $\mathbf{V_{i+1}}$, all processes in $\mathbf{V_i}$ that deliver $\mathbf{V_{i+1}}$ deliver *m* before $\mathbf{V_{i+1}}$.

In other words, between two views all correct processes in the group deliver the same set of messages. View synchronous multicast is an instance of the consensus problem [GS94].

As view synchronous multicast replaces reliable multicast as the foundation upon which the other multicast schemes are built, the FIFO, causal, or totally ordered multicasts in view–synchronous communication also satisfy view synchrony.

### 2.3.4 Replication Strategies

There are different ways to organize the replicas of a replicated object. One generally distinguishes active, semi–active (leader–follower), passive, and coordinator–cohort organization.

*Active* replicas execute in parallel. When a client sends a request to an actively replicated object, all replicas handle the request and reply to the client. If the strong consistency criterion is to be met, this implies that all replicas must behave deterministically, and that they handle the same requests in the same sequence, otherwise, their states and hence their output may diverge. Replicas must therefore adhere to the state machine model [Sch90].

Failures of replicas are masked. Active replicas ensure high availability of the services offered by the replicated component. Because all replicas handle all requests in parallel, a failure causes no synchronization overhead between replicas as for the other replication strategies discussed below.

In *passive* (or *primary–backup*) replication, only one replica (the primary) is active. It is the only replica that handles requests and replies to clients. Before it sends back a reply, it transmits a checkpoint containing its new state and the reply to the other replicas (its backups). If the primary fails, one of the backups becomes the new primary, handling subsequent requests. Failures of backups are masked: a client does not notice them at all. A failure of the primary replica while handling a request necessitates that the client who sent that request re-send it to the new primary. If the failed primary had computed a result for the request and already sent a checkpoint to its backups, the new primary can just return that result. If the failure of the old primary occurred before the checkpoint was transmitted to the backups, the new primary has to re–execute the request.

One of the assets of passive replication is that it can be employed for non–deterministic S–components. [Maz96] describes a solution based on the detection of duplicate messages that can handle passive replication of deterministic CS–components. Classic checkpointing schemes can be viewed as passive replication, where the primary (and only) copy saves the checkpoints to stable storage instead of sending them to the backups. Passive replication offers a higher availability than checkpointing to stable storage because the restart latency after a failure is shorter. Compared to active replication, a failure incurs a higher overhead because some work may have been lost and clients have to re–issue requests. Also note that passive replication cannot be used to mask byzantine failures as there is only one single replica executing: the backups serve only as warm standbys.

*Semi–active* (or *leader–follower*) replication [Pow91] is a hybrid replica organization technique developed within the Delta–4 project to accommodate non–deterministic replicas with an availability nearly as high as in active replication. As in active replication, all replicas receive a request, however, one replica (the leader) plays a special role. Whenever the leader makes a non–deterministic decision, it notifies the other replicas (its followers) of its choice. The followers are then forced to take the same decision. This guarantees that the state evolution in all replicas is the same. In semi–active replication, only the leader replica replies to clients. Semi–active replication will be discussed at considerable length in chapter 6.

*Coordinator–cohort* replication [Bir85] is another hybrid replica organization, very similar to semi–active replication. It has been developed in the context of the Isis toolkit. From the point of view of the communication pattern, it is very similar to passive replication, the only difference being that all replicas receive a request. This makes it possible to mask even failures of the primary replica; the client does not have to re–send a request. However, only the coordinator handles the request and updates the cohort replicas by means of checkpoints. The result is therefore determined by the execution on the coordinator, which may be non–deterministic. If the coordinator fails, one of the cohorts becomes the new coordinator and proceeds with execution from the last checkpoint. Checkpoints therefore must be coordinated with respect to output.

If requests in coordinator–cohort replication are implemented as transactions (see sections 2.4.4 and 5.4), the coordinator can be chosen on a per–request basis. This can be exploited to achieve some load–balancing, but it requires that concurrency control be synchronized between the replicas.

## 2.4 Software Fault Tolerance

Software that can tolerate operational faults of the system components it depends upon (such as the malfunctioning of the underlying hardware) is called “fault–tolerant software”. In contrast, the term “software fault tolerance” is usually understood to mean the ability of software to cope with faults, especially design faults, *within* a component itself. A general, yet concise overview and comparison can be found in [LABK90].

### 2.4.1 Recovery Blocks

Recovery blocks [Ran75] have been introduced as a software structuring mechanism based on backward error recovery in the early 70's. A recovery block consists of one or more alternatives implementing the component's functionality, coupled with an acceptance test that determines whether or not an alternative has functioned correctly. When a recovery block is entered, a recovery point is established by taking a checkpoint of the component's state and the first, primary alternative is executed. If the results from that execution fail the acceptance test or the alternative itself fails, the component's state is rolled back to the checkpoint taken initially and the second alternative is executed. This is repeated until either an alternative passes the acceptance test or there are no more alternatives available and therefore the whole component fails.

Recovery blocks are an inherently application–specific fault tolerance technique. The application must provide the alternatives in a recovery block and it also must implement the acceptance test. Alternatives may implement the same functionality in different ways, or they may try to offer only a degraded functionality when the primary alternative fails the acceptance test. This issue of design diversity is closely related to N–version programming, but there are important differences (see section 2.4.3 below).

Recovery blocks can be nested: alternatives themselves can be implemented as recovery blocks. If a nested recovery block fails, recovery is attempted in the enclosing recovery block by rolling back and then executing the next alternative.

Distributed recovery blocks [Kim95] are an adaptation of the basic recovery block scheme to achieve fault tolerance in a distributed system. A recovery block with two alternatives is replicated on two virtual nodes. When the recovery block is entered, both replicas execute an alternative: one node executes the first alternative as primary alternative while the other node chooses the second alternative as primary. If one alternative fails while the other one succeeds, the failed one is rolled back and the node uses the other alternative to bring its state up to date with respect to the second node. If the primary alternatives succeed on both nodes, it is assumed that they both produce the same results.

### 2.4.2 Conversations

Conversations [HLMR74] are an extension of the recovery block concept to concurrent systems. If concurrently executing processes communicate with each other in the alternatives of recovery blocks, a so–called *domino effect* may occur: if an alternative fails and is rolled back, other alternatives of recovery blocks in other processes also may have to be rolled back. To avoid this very undesirable behavior, the conversation abstraction was proposed as a kind of a coordinated recovery block for cooperating processes. A conversation has the following characteristics:

- Upon entering a conversation a process establishes a checkpoint.
- During a conversation processes can only communicate with other processes in the same conversation.
- If an alternative fails in one process, all processes roll back and attempt to execute the next alternative.
- All processes leave the conversation together.

Conversations, like recovery blocks, can be nested: both support recursive system composition. Forward error recovery can also be used within a conversation to cope with failures. In order to make a consistent recovery of the cooperating processes possible, this necessitates an exception resolution scheme [CR86] because several exceptions may be raised concurrently in different processes participating in the conversation. Conversations with forward error recovery are also known as FT–actions or atomic actions [JC86].

Conversations can be used as a software structure for fault tolerance in distributed systems [MWR98].

### 2.4.3 N–Version Programming

N–version programming (NVP) [AC78] has been developed specifically to cope with design faults. It is defined as the independent development of several (at least two) distinct implementations of the same initial specification of a system component. All the different versions of a component execute a request, possibly concurrently. Their results are then passed to a decision algorithm that tries to find a consensus value from all the results. This consensus value is then the result produced by the system component. The decision algorithm is application–specific. Whereas the acceptance test of a recovery block is applied to the single result of an alternative to determine success or failure, the decision algorithm in NVP uses the results of all versions to arrive at a single consensus result.

The key point is the independence of the implementations, which shall avoid that implementations share common design errors [Avi85]. NVP has evolved into an elaborate design paradigm. Each version should be implemented by a separate team; teams ideally should not communicate with each other to avoid that one team's ideas influence another team and both end up implementing the common specification by the same algorithm, which would increase the likelihood that they committed the same design errors. Different implementations may even be built using different hardware and development tools to avoid common errors creep in, for instance due to a faulty compiler.

The weak point of NVP is the decision algorithm. It can be far from trivial to specify such an algorithm, furthermore, this specification may depend heavily upon the application's semantics. Also, the decision algorithm itself must be fault–tolerant if the system component using NVP is to be fault–tolerant.

### 2.4.4 Transactions

Transactions [Gra78] are a well–known concept for structuring an application to provide data consistency in a concurrent environment and in the presence of failures. In the world of database applications, a transaction is an execution that is characterized by the famous ACID properties [HR83]:

- ***A****tomicity:* a transaction gives an "all or nothing" guarantee: either all its state changes are performed, or none are, even in the presence of failures (failure atomicity).
- ***C****onsistency:* the execution of a transaction starting in a consistent state will produce another consistent state.
- ***I****solation:* concurrent transactions are isolated from each other; their execution has the same effect as some serial execution order.
- ***D****urability:* once a transaction has terminated successfully ("*committed*"), its effects are permanent even when a failure occurs.

From a programmer's point of view, a transaction is a sequence of statements encapsulated between a "beginning of transaction" statement (BOT) and an "end of transaction" statement (EOT). A transaction can either *commit* at EOT if the execution was successful or *abort* if some error occurred. If it is committed, any state change it made becomes visible to other transactions. If it is aborted, it has no effect at all; any state changes that it might already (tentatively) have made must be undone.

Transactions can be nested [Mos81]. Nested transactions enhance the "flat" transaction model by providing modular composition of transactions, dealing with intra–transaction concurrency, and offering a way to handle partial failures. They are particularly well suited for applications in distributed systems.

The term "transaction" has a strong connotation of being limited to database systems. In the domain of software fault tolerance, the term "atomic action" is usually used, though this latter term has been used somewhat inconsistently lately. Some researchers use it to denote transactions, while others use it as a synonym for conversations. Although these two approaches can be seen as different views of the same problem [SMR93], I do not use the term "atomic action" in this thesis to avoid possible confusions.

For a detailed discussion of transactions, the reader is referred to section 5.4.

# Chapter 3:
# Distributed Systems in Ada 95

Ada 95 is a revised and much improved version of the “classical” Ada programming language developed originally for the United States Department of Defense to match their requirements for a modern, safe, and efficient structured programming language. Classical Ada was codified as an ANSI standard in 1983 and is therefore sometimes called “Ada 83”; an equivalent ISO standard was ratified in 1987. The successor language Ada 95 is defined by ISO standard ISO/IEC 8652:1995 [ISO95]. For an overview of the history of Ada, see the rationale [Bar95].

In order to lay a solid foundation for the discussion in the following chapters, I give a very brief overview of the most important new features in Ada 95 here, with an emphasis on its provisions for the development of distributed systems.

## 3.1 Ada 95

Ada 95 improves over the original definition of Ada 83 in several areas, including the following:

- Addition of language constructs for object–oriented and incremental application development (“tagged types” and “child packages”).
- Improvements in the area of tasking: a new construct, called “protected type”, implements passive entities that may be accessed concurrently by several tasks (monitors).
- A distribution model based on remote procedure calls is standardized.

The last point is explained in detail below (see section 3.2 on page 31). In the remainder of this section I will present the two other novelties, tagged and protected types, for the following chapters assume the reader is familiar with them. (For a complete description of all the new features in Ada 95, see the language standard [ISO95] and the rationale [Bar95].)

### 3.1.1 Object–Oriented Programming

Object–oriented programming in Ada 95 is based on the concepts of derivation classes formed by type extension [Wir88a] and of single inheritance.

Type extension works by refining an existing record type by adding new components or operations, or by modifying existing operations. Contrary to e.g. Oberon–2 [MW91], not all record types can be extended: only *tagged* types can. (This language design choice was motivated mainly by a concern for efficiency.) A tagged type is a record type defined with the keyword `tagged`, or a type derived from such a tagged record type. Fig. 3.1 shows a simple tagged type with a few extensions. The derived types inherit all their ancestors' components and primitive operations[1]. New components may be added with each derivation, inherited primitive operations may be overridden if the inherited behavior is not appropriate for the derived type, and new primitive operations may be added.

```
with Canvases; use Canvases;
package Shapes is

  type Shape is
    abstract tagged null record;

  procedure Draw
    (S      : in     Shape;
     Canvas : access Canvas'Class)
    is abstract;

end Shapes;

package Shapes.Circles is

  type Circle is new Shape with
    record
      -- Added components
      Center : Point;
      Radius : Float;
    end record;

  procedure Draw
    (C      : in     Circle;
     Canvas : access Canvas'Class);
  -- Inherited abstract primitive
  -- operation, must be overridden.

  function Radius
    (C : in Circle) return Float;
  -- New primitive operation;

end Shapes.Circles;

package Shapes.Rectangles is

  type Rectangle is new Shape with
    record
      Top_Left     : Point;
      Bottom_Right : Point;
    end record;

  procedure Draw
    (R      : in     Rectangle;
     Canvas : access Canvas'Class);
  -- Inherited and overridden.

  function Width
    (R : in Rectangle) return Float;
  -- New primitive operation;

end Shapes.Rectangles;
```

**Fig. 3.1:** A Tagged Type Hierarchy

1. In other object–oriented languages, the term "method" is often used.

A particularity of the object–oriented concepts of Ada 95 is class–wide programming. It makes explicit the dynamic polymorphism that in other object–oriented languages often is inherent. For each tagged type `T` there is a corresponding class–wide type `T'Class`, comprising the whole tree of types derived directly or indirectly from `T`, including `T` itself. Values of such a class–wide type can hold any value of any of the types derived from `T`. In particular, access–to–class–wide types may be used to reference any value of any type in the derivation class, enabling a programmer to code e.g. heterogeneous collections.

Another common use of class–wide types is for dispatching calls, i.e., calls to primitive operations where the target operation is determined *at run time* depending on the dynamic type of the value of a class–wide type (i.e., depending on its tag). This is different from most other object–oriented programming languages, where either all method invocations are dispatching (e.g., in Java), or where it must be specified at the declaration of the method whether or not calls will be dispatching (e.g., the `virtual` methods in C++). In Ada 95, the programmer can decide at each call whether or not it should dispatch. If the controlling operand (actual parameter) of a call to a primitive operation has a class–wide type, the call is dispatched, otherwise, the primitive operation of the type of the controlling operand is invoked. The difference is illustrated by the code fragment in fig. 3.2.

```
type Shape_Ref is
  access all Shape'Class;

...

declare
  A_Shape  : Shape_Ref := ...;
  A_Canvas : aliased Canvas;
begin
  Draw (A_Shape.all,
        A_Canvas'Access);
  -- This call dispatches on the
  -- actual type of the object
  -- 'A_Shape' references -- this
  -- might be either 'Circle' or
  -- 'Rectangle' in this example.
end;

declare
  A_Circle : Circle := ...;
  A_Canvas : aliased Canvas;
begin
  Draw (A_Circle, A_Canvas'Access);
  -- This call is not dispatching:
  -- the actual parameter is not of
  -- a class-wide type.
end;
```

**Fig. 3.2:** Illustrating Dispatching Calls

Dispatching in Ada 95 is *safe* — there always exists a primitive operation to dispatch to at run time; it is not possible to write a program that would dispatch to a non–existing method (as it may happen e.g. in Smalltalk): the language rules do not allow this, and hence such errors will be caught by the compiler.

Dispatching occurs only on primitive operations; it is never controlled by formal parameters with class–wide types. A call to a subprogram that is not a primitive operation but has formal parameters with class–wide types never dispatches. Such subprograms are called *class–wide* operations. Any actual parameter that has a type in the given derivation class may be passed in place of a formal class–wide parameter.

Finally, Ada 95 provides *abstract* tagged types that may have abstract primitive operations. An abstract type defines characteristics common to all types derived from it; its abstract primitive operations are inherited and form a specification that all types derived from it have to adhere to. An abstract type may also have concrete primitive operations — this is useful to express default behavior for all types derived from the abstract type. A concrete type derived from an abstract type must supply implementations of the inherited abstract operations.

### 3.1.2 Controlled Types

Controlled types [ISO95, 7.6] have been introduced in Ada 95 to facilitate resource management within abstract data types while preserving the abstraction. To make a type controlled, it must be derived from one of two standard abstract tagged types, `Ada.Finalization.Controlled` or `Ada.Finalization.Limited_Controlled`. The controlled type then inherits the following operations (which are invoked automatically) from these root types:

- `Initialize` — is invoked whenever an object of the controlled type is created (and there is no explicit initialization).
- `Finalize` — is called just before a controlled object is destroyed, i.e., goes out of scope, is deallocated using an instantiation of `Ada.Unchecked_Deallocation`, or is overwritten during an assignment.
- `Adjust` — is invoked for the target object in an assignment just after it has been overwritten.

(Type `Limited_Controlled` has of course no primitive operation `Adjust`, since objects of limited types cannot be copied.) The default implementations of these three primitive operations do nothing at all. By overriding the inherited versions in the derived type, the application developer can precisely control object creation, destruction, and assignment. This can be used for instance to implement automatic storage management for an abstraction at the application level without having to clutter its interface.

### 3.1.3 Protected Types

One of the highlights of Ada has always been its integrated model for structuring concurrent computations using tasks and the rendezvous concept. The rendezvous model presents an abstraction from low–level synchronization primitives such as signals or semaphores.

However, experience with Ada over the years has shown that the rendezvous alone is not entirely sufficient to express synchronization in a convenient way. One problem is that rendezvous basically is control–flow oriented and doesn't lend itself easily to data synchronization. Shared data often had to be encapsulated within extra tasks in Ada 83, complicating the programs and leading to poor performance.

Ada 95 therefore introduced the concept of protected types. A protected type encapsulates some data items and offers synchronized access to this data through access routines. Access routines may be read–only (expressed as functions) or read–write (procedures and entries). The language guarantees mutual exclusion between all accesses to a protected object with the usual semantics of multiple readers or a single writer. In this respect a protected object is equivalent to a monitor [BH73]. A simple protected object encapsulating some data item that is to be accessed by several tasks might be written as shown in fig. 3.3 below.

```
protected type Shared_Data is

  procedure Set (Val: in Data_Type);

  function  Get return Data_Type;

private

  Item : Data_Type;

end Shared_Data;
```

```
protected body Shared_Data is

  procedure Set
    (Val: in Data_Type) is
  begin
    Item := Val;
  end Set;

  function Get return Data_Type is
  begin
    return Item;
  end Get;

end Shared_Data;
```

**Fig. 3.3:** Protected Type for Mutual Exclusion

C. A. R. Hoare extended this basic monitor concept in [Hoa74] with so–called "condition variables" that could be declared inside a monitor. These basically are signals. A procedure of a monitor may suspend itself by waiting for the condition to become true. Another procedure of the monitor will signal on the condition variable to indicate that the condition has become true. In Hoare's scheme, the `wait` operation relinquishes exclusion (to allow some other task to enter the monitor on the other procedure), and a `signal` operation immediately resumes a waiting task if there is one, making the signalling task leave the monitor.

The protected types of Ada 95 offer a similar feature, albeit in a more versatile way and at a higher level of abstraction. It has been noted that condition variables suffer from the same drawbacks as simple semaphores, notably that their correct use is not always easy and that the use of explicit `wait` and `signal` operations is just as error–prone as the use of `P()` and `V()` operations on a semaphore. Ada 95 solves this problem by introducing barriers: an `entry` of a protected type may have an associated *barrier*, a boolean expression that is (usually) expressed in terms of the state encapsulated in the protected type. A task calling an entry can only enter the protected object if the barrier is true; if not, the task is suspended until the barrier becomes true. Other tasks may enter the protected object through other entries or procedures, though. The run–time system checks at the end of each procedure or entry call whether any barriers on which tasks are waiting have become true, and if so, resumes a waiting task, letting it execute the entry. An example of a protected type implementing a bounded buffer is given in fig. 3.4.

```
Max_Elems : constant Natural := ...;

type Count_Type is
  new Natural range 0 .. Max_Elems;

type Index_Type is
  new Natural range 1 .. Max_Elems;

type Buffer_Type is
  array (Index_Type) of Data_Type;

protected type Bounded_Buffer is

  entry Append (Val: in  Data_Type);

  entry Remove (Val: out Data_Type);

private

  Buffer      : Buffer_Type;
  First, Last : Index_Type := 1;
  Nof_Elems   : Count_Type := 0;

end Bounded_Buffer;

protected body Bounded_Buffer is

  entry Append (Val: in Data_Type)
    when Nof_Elems < Max_Elems is
  begin
    Buffer (Last) := Val;
    Last := Last mod Max_Elems +1;
    Nof_Elems := Nof_Elems +1;
  end Append;

  entry Remove (Val: out Data_Type)
    when Nof_Elems > 0 is
  begin
    Val := Buffer (First);
    First := First mod Max_Elems +1;
    Nof_Elems := Nof_Elems -1;
  end Remove;

end Bounded_Buffer;
```

**Fig. 3.4:** Protected Type with Entries

Ada 95 enhances this model of monitors even further by offering a few more language constructs:

- An attribute `'Count` may be applied within a protected object to one of its entries to obtain the number of tasks waiting on its barrier to become true. ("`Entry'Count > 0`" corresponds to Hoare's "`condition.queue`" predicate.)
- An attribute `'Caller` may be used within an entry to get the task identifier of the calling task.
- A `requeue` statement allows a programmer to requeue an entry call on the same or some other entry (in the latter case, the parameter profile must match).

These features greatly extend the expressive power of protected types. In particular, the `requeue` statement makes protected types capable of implementing preference control schemes at the application level; an example would be a resource allocation server that granted satisfiable requests but queued currently unsatisfiable requests for later servicing. In Ada 83, there was no satisfactory way to implement such servers.

### 3.1.4 Asynchronous Transfer of Control

Asynchronous transfer of control (ATC) is another important feature of Ada 95, allowing one task to signal another task without having to use a (synchronous) rendezvous. The language provides an asynchronous `select` statement of the form given in fig. 3.5.

```
Asynchronous_Select :=
  select
    Triggering_Alternative
  then abort
    Abortable_Part
  end select;

Triggering_Alternative :=
  Triggering_Statement
  [Sequence_Of_Statements]

Abortable_Part¹ :=
  Sequence_Of_Statements

Triggering_Statement :=
  Entry_Call_Statement |
  Delay_Statement
```

1. Must not contain `accept` statements!

**Fig. 3.5:** Syntax for Asynchronous `select` Statements

Using the asynchronous `select` statement, a task may alter its flow of control depending upon the asynchronous occurrence of external events. Applications of this include for instance mode changes in real–time systems, user interrupts or time–outs aborting lengthy computations, or error recovery [BW95].

The abortable part of an asynchronous `select` statement is started if the triggering statement is a delay that hasn't yet expired or an entry call that is queued (or later requeued using `requeue ... with abort`). If then the triggering statement completes before the abortable part, the latter is aborted; otherwise, the triggering statement is aborted. See [ISO95, 9.7.4] for the complete rules (which are quite subtle).

If abortion were allowed to occur at any moment during the execution of the abortable part or of the trigger, it would be nearly impossible to maintain the consistency of the state accessed in an asynchronous select statement. The standard therefore defines a number of language constructs that defer abortions [ISO95, 9.8(6–11)], most notably protected actions and the `Initialize`, `Finalize`, and `Adjust` operations of controlled types. During these *abort–deferred* regions, abortions cannot occur: they are delayed until the abort–deferred region is left. I will discuss these topics in more detail on page 83 in section 6.4.3.

## 3.2 Annex E: Distributed Systems

Ada 95 is the first programming language that defines a precise model for the development and structuring of distributed applications *in the language standard*. In this section, I present an overview of the relevant annex E of [ISO95].

### 3.2.1 The System Model

In an abstract way, a distributed application can be viewed as a collection of cooperating entities or program fragments (so–called "virtual nodes") that themselves are indivisible as far as distribution is concerned and that are distributed on various computers (or "physical nodes") in a network, over which they communicate with each other. This is precisely the view Ada 95 adopts, too. A virtual node in Ada 95 is called a *partition*[1]. A partition 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 as belonging to the interface (see section 3.2.2 below).

Ada 95 distinguishes *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. Direct remote data access between active partitions does not exist in Ada 95.

Partitions are semi–autonomous: while they only make sense as part of a larger distributed application and thus are intrinsically bound to cooperate with the other partitions, they evolve independently from each other in matters concerning tasking, time, I/O, and so on. The language standard does not require a distributed run–time support that might offer a common base for such services. A direct consequence of this is that all tasks are local to a partition; they're not visible across partitions and hence there is no remote rendezvous in distributed Ada 95.

The configuration of distributed applications is beyond the language standard. Except for the restrictions mentioned below in subsection 3.2.2, it covers neither the assignment of library units to partitions nor the allocation of partitions to physical nodes.

### 3.2.2 Partitions and Packages

Ada 95 has adopted a mixture of the pre– and post–partitioning approaches to the development of distributed systems. A partition is built by assembling the various packages that it should contain, *after* the application has been written and compiled. Partitioning can be done statically or even dynamically. However, the developer has to foresee the possible partitionings from the onset and has to provide for them: the interfaces between partitions must be defined in advance, *before* the partitions are built. This mixed approach yields the bene-

1. Not to be confused with a *network partition*, i.e. the rupture of communication links such that a formerly connected network is split in several parts that cannot communicate with each other anymore.

fits of both: because partition interfaces are known at compile time, the compiler can apply semantic checks even across partitions, thereby maintaining the type safety of the whole application; and because partitions themselves are defined only after the compilation phase, an application can be partitioned in different ways without recompilation, subject to the constraints given by the “seams” (remote interfaces) provided by the developer.

Ada 95 offers several categorization pragmas, which form a hierarchy, to define the interfaces of the partitions of distributed applications. These pragmas must be used to class all library units of a distributed application in one of the following five categories:

*Pure* packages are preelaborable stateless library units with the additional restriction that they must not declare library–level named access types. Each partition that needs a certain `Pure` library unit contains its own copy of it: as it is stateless, inconsistencies cannot arise. Logically, a pure library unit behaves as if it existed only once. Note that package `Standard` in particular is a `Pure` package, which guarantees that one can use the standard types in communication between active partitions.

*Shared_Passive* library units also must be preelaborable and must not globally declare task types, protected types with entries, or access types to class–wide types. State (global variables and protected objects without entries) declared in the specification of a `Shared_Passive` package are directly accessible by active partitions. The constraints placed upon these library units make sure that such accesses can never trigger a remote call to another active partition (e.g., through dispatching). A `Shared_Passive` library unit can be assigned to at most one partition.

The specifications of *Remote_Types* library units (“RT units”) also must be preelaborable. In addition, they must not declare variables in their interface. However, they may declare access–to–subprogram types and access types to class–wide limited private types. Any access type declared in the interface of a `Remote_Types` unit becomes a *remote access type*. The restrictions make sure that any dereference of a value of such a remote access type can only occur in the context of a remote call (in the case of a remote access–to–subprogram type) or a remote dispatching call (remote access–to–class–wide type). Thus, remote accesses do *not* provide remote data access, but serve as a means for late binding of remote calls. (See also section 3.2.4 below.)

Pragma *Remote_Call_Interface* is used to designate library units (“RCI units”) declaring remotely callable subprograms. Any subprogram declared in the visible part of the interface of an RCI unit can be called remotely from some other active partition. The body of an RCI unit must be assigned to exactly one active partition. RCI unit interfaces may not declare state (i.e., variables) or limited types: this excludes also tasks and task types.

Finally, normal library units (without any of the above four pragmas) are completely unrestricted. Each partition that needs a normal library unit has its own copy of it. Any entity declared in a normal library unit is local to the partition it is contained in and cannot be accessed remotely; in fact, each copy and whatever entities it declares are considered distinct. Since there is no distributed run–time support, the state of such library units evolves independently from that of their counterparts that might exist in other partitions.

Categorized library units form a hierarchy, in the order they are presented above. A `Shared_Passive` library unit may depend only upon other `Shared_Passive` units or upon `Pure` units. The specification of a `Remote_Types` unit may depend only upon other `Remote_Types` units, or upon `Shared_Passive` or `Pure` units; the body of an RT unit is unconstrained in this respect. Units categorized as `Remote_Call_Interface` are similar: their specification must not depend upon normal library units, whereas there's no such restriction on their bodies.

The interface of a passive partition is given by the union of the interfaces of all the `Pure` and `Shared_Passive` library units it contains. The interface of an active partition is defined by the union of the interfaces of all `Pure`, `Remote_Types` and `Remote_Call_Interface` library units it contains; normal packages contained in active partitions are not visible across partitions.

### 3.2.3 Remote Procedure Calls

Communication between active partitions is based solely on the concept of remote procedure calls (RPC, [BN84]), a communication abstraction for interpartition communication built on top of simple message passing. A remote procedure call in Ada 95 is nearly transparent to the application developer: excepting the categorization pragmas, both the implementation of a remotely callable subprogram as well as its call do not differ at all from a local subroutine.

A remote procedure call can be decomposed into five distinct phases, as shown in fig. 3.6 below. The five basic phases of an RPC are:

1. In order to send the arguments of the call over the network, they must be flattened into a single stream of bytes. This process is called *marshaling* and is done transparently for the application in the *caller's stub* version of the remotely callable subprogram. The caller's stub also adds some identification of the routine to be invoked to this stream of bytes. The run–time support then sends an *RPC request message* to the partition the implementation of the desired subprogram resides on.
2. On the receiving side, the run–time support gets the information which subprogram is to be called from the request message and invokes the corresponding *receiver's stub* procedure. The receiver's stub then reconstructs an internal representation of the arguments from the byte stream in the request message (this is called *unmarshaling*) and locally invokes the correct subprogram.
3. The remotely callable subprogram executes.
4. The receiver's stub marshals all the values returned from this subprogram, i.e. `inout`– and `out`–parameters, function results, or even the identity of an exception, if one was

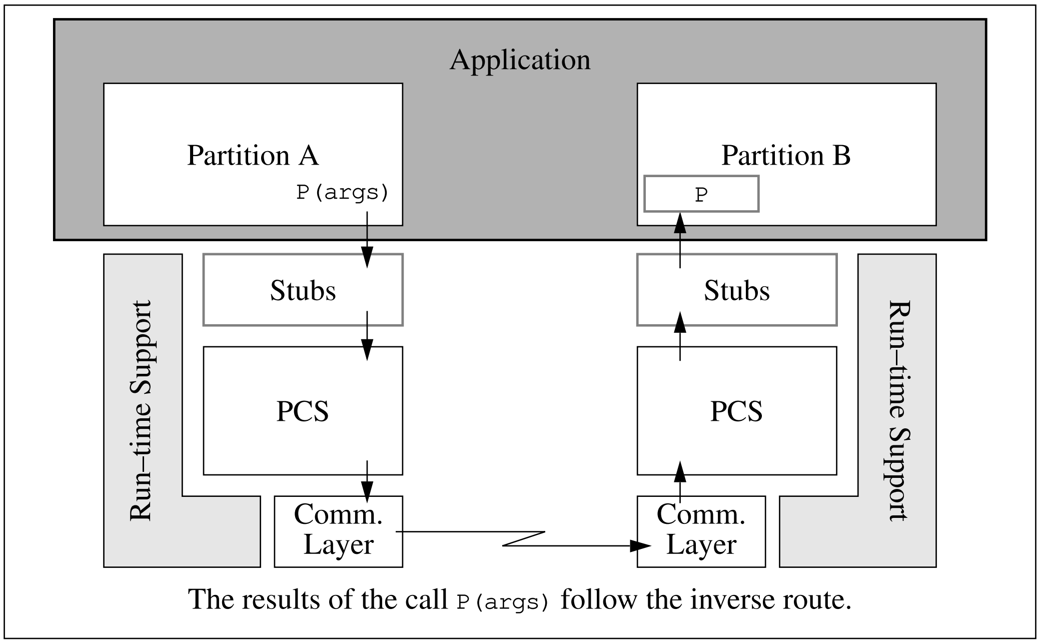


**Fig. 3.6:** Schematic View of a Remote Procedure Call

raised in step 3. The run–time support then sends an *RPC answer message* back to the calling partition, containing the results of the call (if any).

5. On the calling partition, the run–time support passes this answer message back to the caller's stub, which then unmarshals the results and passes them back (possibly including the raising of an exception) to the original point of call in the application.

The above describes the basic remote procedure call in Ada 95. This is called a *synchronous* or *blocking* RPC, because the caller proceeds only after having received the answer message. The language standard also offers so–called *asynchronous* or *non–blocking* RPCs, in which steps 4 and 5 of the above list may be omitted and the caller may continue its execution after having sent the request message, possibly before the remotely called subprogram has finished its execution[1]. Asynchronous RPCs are a way to provide a type–safe interface to message passing in Ada 95. Asynchronously called routines cannot return results and may have only parameters of mode `in`. Any exceptions they might raise are lost on the caller's side; they are not propagated to the calling partition. Whether a certain remotely callable subprogram is to be called asynchronously is defined statically by applying pragma `Asynchronous` to the declaration of the subprogram.

As I mentioned above, RPCs are highly transparent: the callers' and receivers' stubs are generated automatically by the compiler or by a partitioning tool. The marshaling and unmarshaling process uses the standard `Stream` facility of Ada 95 and is done via the

1. Note that this formulation allows asynchronous RPCs to be implemented as synchronous ones: the only difference concerns the propagation of exceptions.

attributes `'Write` and `'Read` or `'Output` and `'Input`. An application having special needs regarding the conversion of certain types into byte streams may provide its own implementations for these attributes; thus marshaling and unmarshaling are customizable. The message passing protocol underlying a remote procedure call and described above is completely hidden from the application in the PCS (Partition Communication Subsystem, see section 3.2.5), part of the run–time support. The PCS is invoked by the stubs through a standardized interface in package `System.RPC`.

`Remote_Types` packages may declare remote access–to–subprogram types in the visible parts of their interfaces. These make dynamic remote calls possible: a variable on one partition may contain a reference to a remotely callable subprogram on some other partition. Again, distribution is transparent: an indirect call through a remote access–to–subprogram differs in no way from a local indirect call. The actual call follows again the scheme shown in fig. 3.6 above. The only difference is that the destination of the call is not known statically but only at run time.

### 3.2.4 Distributed Objects

The distribution model of Ada 95 also covers the realm of object–oriented programming. The language standard uses both the new OO features and the new distribution features to define a distributed object model, which adds great flexibility to both areas.

An object in Ada 95 always resides on the partition it was created on; despite the designation "distributed object", it cannot be passed from one partition to another one. However, it *is* possible to have remote accesses to objects that reside on other partitions. In conjunction with dispatching calls; such remote references provide *remote dispatching* as a further means to dynamically designate the destination of a remote call.

A remote access–to–class–wide type may designate only class–wide limited private types. This restriction ensures that

- the object that is referenced cannot migrate[1], and
- other partitions cannot access the object's data directly.

The language standard [ISO95, E.2.2.(16)] states that a value of a remote access–to–class–wide type can only be dereferenced to designate the controlling operand of a dispatching call[2]. This is the only way other partitions can access the referenced object.

Distributed object types are usually declared in `Remote_Types` packages, which can and usually do exist in several copies in a distributed application: each partition that references a `Remote_Types` package contains its own copy of it. Still, the types declared by these

1. If an implementation provided support for the migration of whole partitions, the objects contained in them would of course migrate with them. However, an object cannot migrate separately.
2. Note that this forbids remote calls to class–wide operations, the reason being that the controlling object might reside on some other partition than the operation, or that the operation itself might exist twice in such a case, and it wouldn't be clear which one to call.

physically distinct copies are considered one and the same. It is this seemingly awkward definition which gives `Remote_Types` packages their power and which truly extends the object–oriented programming paradigm to distribution.

It is possible to have objects of the same derivation class (or even of the same tagged type) on two different partitions. A third partition may obtain remote references to these objects in the form of values of a remote access–to–class–wide type. This third partition can then use dispatching calls to invoke primitive operations on the partition that created the object without having to know explicitly *which* partition that is. In a remote dispatching call, the caller determines the receiver partition from the remote access value in a transparent way. The dispatching happens on the receiving partition, where the object resides. Note that the dispatching *must* happen at the receiver's side, as the class of a tagged type declared in a `Remote_Types` package may well be extended in several other `Remote_Types` or RCI packages, and the set of such packages may be different on each partition. Furthermore, the derived type of the actual object referenced by the remote access may not even exist on the caller's side, and hence dispatching can only be done at the receiver. These steps aside, a remote dispatching call follows exactly the steps given in fig. 3.6 for simple remote procedure calls.

### 3.2.5 The Partition Communication Subsystem

The Partition Communication Subsystem (PCS) is that part of the run–time support of Ada 95 that implements the remote procedure call abstraction. The standard does not specify how this is to be accomplished, but it does specify a standardized interface to the PCS that the compiler and the callers' and receivers' stubs shall use. This interface is given by the declaration of the standard package `System.RPC`.

Each partition is identified by a partition ID. To make a remote procedure call, the caller's stub calls procedure `System.RPC.Do_RPC` giving the partition ID of the destination partition and the marshaled parameters of the remote call in the form of a `Stream` as parameters. A second `Stream` is used for passing back the marshaled results from the run–time support to the stub once the RPC has successfully completed. `Do_RPC` then implements the steps given in section 3.2.3 above to perform the remote call. For an asynchronous RPC, the caller's stub invokes `System.RPC.Do_APC`, which makes an asynchronous remote call.

On the receiver's side, it is left undefined how exactly the receiver's stub is invoked. The standard assumes that there is an RPC receiver subprogram that receives all incoming streams. It is called by the run–time support and has the task of calling the correct stub. Note that this RPC receiver is completely transparent to the application, it is somehow provided by the implementation [ISO95, E.5(21)]. The standard requires that this RPC receiver be re–entrant [ibid, E.5(24)] and strongly suggests that the PCS handle RPC requests concurrently [ibid, E.5(25, 28)], i.e. that it handle each RPC request in a separate task.

### 3.2.6 Fault Tolerance

The language standard for Ada 95 does not require distributed applications to be fault–tolerant at all. The only provision made is the predefined exception `Communication_Error` that may be raised on the partition initiating a remote call when the destination partition is inaccessible. Apart from that, [ISO95, E.1(12)] specifically states that an implementation may allow replication of partitions, but this is only an implementation permission, not a requirement.

### 3.2.7 Open Computing and Ada 95

What are the limits within which the above standardized model of distributed computing in Ada 95 can be used?

Annex E of the language standard is open enough to support heterogeneous distributed systems, although it allows implementations to restrict themselves to homogeneous systems [ISO95, E(6)]. Using Ada 95 for distributed applications in a heterogeneous environment has some pitfalls, though. It seems as if part of the language definition was made with the idea of homogeneous distributed systems only. [ISO95, E.2(13)] specifies that all remote types (i.e., library–level types declared in `Pure` or `Shared_Passive` library units, or in the declarations of `Remote_Types` or `Remote_Call_Interface` library units) must have the same representation on all partitions that use the unit. The precondition to using Ada 95 to program an application that is to be executed in a heterogeneous environment is therefore to find a development system that represents all remote types identically on all platforms that are used. Note that this also means that all standard types must be represented the same because package `Standard` is categorized as `Pure`.

This is a severe restriction that is — in my opinion — not necessary. I am not aware of any reasons that would support this requirement; even remote access types and remote dispatching can be implemented perfectly well with different representations on different partitions (see section 8.8). It would suffice if the standard required that all remote types have identical *declarations* on all partitions, regardless of the precise representation of these types. This guarantees that the abstract properties of all types, even those declared in the standard libraries, are identical on all partitions. As long as all partitions use a common, platform–independent encoding of remote types, any necessary conversions can be done automatically and transparently for the application when values are marshaled or unmarshaled in the stream attributes. A comment on this has been submitted as AI–208 to the ARG (Ada Rapporteur Group), but has not yet been answered definitively.

Interoperability between different Partition Communication Subsystems is not covered at all by the Ada 95 language standard, as it does not codify the low–level communication protocols that shall be used to implement the high–level abstraction of remote procedure calls. It is in general not even possible to freely choose the PCS one wants to use; one normally has to use the one supplied by the compiler vendor. The problem is that the specification of the interface of the PCS (`System.RPC`) is geared towards a particular implementation model. The `Do_RPC` and `Do_APC` routines get the routing information for an RPC request message (i.e., which partition to send it to) by means of the `Partition_ID` parameter. The RPC stub generated by the compiler must supply this information. It follows that the stubs must know the destination partition's ID up front, which in turn implies that partition IDs be assigned statically at the time the distributed application is configured. (Inside the implementation of remote calls — and the stubs are part of this — one cannot use the attribute `'Partition_ID` to obtain the partition ID.) Not all development systems may want to use this fairly restricted model. For instance, if partition IDs are to be assigned dynamically during the elaboration of the partitions, the run–time support must include some kind of registry for partition IDs. The RPC stubs that the compiler generates must then query this partition ID server before calling `Do_RPC` or `Do_APC`. As a consequence, only a PCS that adheres to a particular compiler's conventions about this partition ID server can be used with that compiler.

It might have been a good idea to include a more elaborate interface specification for the PCS in the standard and to require that any implementation use *only* this interface to implement remote calls[1]. In this way, one would have had the freedom to use the PCS of one's choice together with one's preferred compiler. As it is, the compiler and the PCS are usually tightly coupled to one another.

### 3.2.8 GNAT

GNAT [SB94] originally stood for "GNU New York Ada Translator"[2]. It is a development system integrated into the well–known GNU environment, and it is distributed freely under the GNU public license. It consists mainly of an Ada 95 front end for the GNU `gcc` compiler system, a full implementation of the standard libraries and the standard's annexes, special linking utilities for Ada 95, a make utility that knows about dependencies between Ada library units, plus a partitioning and configuration tool called `gnatdist`.

With GNAT, a distributed application is partitioned statically using `gnatdist`. This tool accepts as input a description of the desired partitioning written in a special–purpose configuration language. From this description, it compiles the needed library units and assembles them into partitions. This partitioning is static. If a different partitioning of an application is desired, `gnatdist` must be re–run with a new configuration file. Partitions can also be allocated to physical nodes by means of this configuration language, but this can also be done separately and differently for each run of the application.

The implementation described in chapter 8 of this thesis is done in the context of GNAT, and I will give a more detailed account of some relevant aspects of this system there.

1. Note that [ISO95, E.5(1)] omits this "only"!
2. The official position from ACT, who are developing and maintaining GNAT today, is now that GNAT means just "GNAT"...

# Part II

# Replication in Ada 95

# Chapter 4:
# Prelude

In this second part of my thesis, I discuss fault tolerance by replication for distributed Ada 95 [ISO95] programs. This introductory chapter states the goals I wanted to achieve, defines the system model I assume, and explains the fundamental problems replication in Ada 95 entails.

## 4.1 Goals

The goal of this work is to render applications written in Ada 95 fault–tolerant using replication. Following the spirit of the Ada 95 language standard, which tries to keep the semantic differences between centralized and distributed execution at a minimum, replication is to be offered in a transparent way. Ideally, the application is not aware of the fact that some of its partitions are replicated, and no special–purpose code should have to be written. I view replication basically as a configuration issue: as I strive to maintain the usual semantics of the programming language, an application should not have to be rewritten to accommodate replicated partitions. This view is backed by the language standard itself. [ISO95, E.1(12)] states that

> *"An implementation may allow separate copies of an active 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."*

This implies that replication can be handled entirely within the configuration phase of a distributed application, after it has been programmed and compiled. As explained in the introduction in chapter 1, this means that replication should maintain both *replica transparency* (i.e., transparency towards the application–level of the replicated object itself) and also *replication transparency*, i.e. transparency towards the other objects in the system.

In practice, the noble goal of complete transparency may be impossible to achieve. Nevertheless, it shall be the lodestar for this work. Any non–transparencies are to be kept at an absolute minimum, and should certainly not interfere with the defined semantics of the language.

## 4.2 The System Model

Throughout this thesis I will assume the following system model: A distributed application executes on an *asynchronous* distributed system. The virtual nodes of the distributed application do not share memory, but communicate only by message passing over the network through *reliable channels*. Nodes are subject only to *crash failures* (fail–silent behavior). Fault tolerance is to be achieved for operational hardware faults; design faults and software fault tolerance are not considered.

I assume that the asynchronous distributed system is augmented by a group communication layer offering view synchronous group communication and group membership information.

Note that I do *not* consider real–time systems, given the assumption of an asynchronous distributed system. The possible existence of passive partitions is disregarded: their intended use is restricted to either physically shared memory, in which case measures to ensure fault tolerance outside the scope of this thesis have to be taken, or to distributed shared memory implementations, which I assume are responsible themselves for providing a fault–tolerant distributed shared memory.

The distributed systems considered may possibly be heterogeneous, i.e., they may be composed of different types of physical nodes.

## 4.3 Replication Units

The partition is the unit of distribution in Ada 95. But what should be its unit of replication? It seems obvious that both should have the same granularity; nevertheless there are a few alternatives, which I discuss in this section.

### 4.3.1 Protected Objects

Ledru proposed in [Led95] a scheme to replicate protected objects. Protected objects that were to be replicated would be declared in packages marked with a new categorization pragma `Shared_Active`. These packages could be assigned to several partitions. Access to the protected objects would be synchronized amongst all the replicas of an object. The main advantage over the Shared_Passive packages defined in [ISO95] is that protected objects might even have conditional entries. However, it is clear that this approach cannot offer fault tolerance in our sense. It is intended only as a way to implement a distributed shared memory system and to improve the throughput of systems where read accesses to protected objects occur more often than write accesses. By replicating the object on each partition that wants to access it, read accesses can be handled locally, thus improving the overall performance. Write accesses incur a coordination overhead between the replicas, though. Ledru's proposition uses a centralized sequencer to order accesses to a protected object. The client packages using the shared state are not replicated in this approach.

The protected object per se is too fine grained an abstraction to provide replication for fault tolerance. It only allows to replicate state — the code using that state is not replicated. Upon a node failure, the application would thus be left with some copy of the state, but possibly crucial other parts of it would still be missing! Also, some restrictions are placed upon such replicated protected objects, e.g., they must not be used as triggering statement in an asynchronous transfer of control (ATC). The proposed approach is not fault–tolerant: if the node running the sequencer fails, all copies of the protected objects in Shared_Active packages become inaccessible. These drawbacks notwithstanding, it is an interesting extension of the Shared_Passive concept of Ada 95.

### 4.3.2 Types

In [WB96], Wellings and Burns proposed an approach to replication based on types, inspired by both the controlled types and the facilities provided for distributed objects in annex E of the language standard.

A new package similar to package Ada.Finalization would provide coordination between objects of that type in a transparent manner. This new package is shown in fig. 4.1 below. Type Broadcast_Data is intended to give access to a simple broadcast facility without any delivery guarantees, Reliable_Broadcast_Data should implement the semantics of reliable multicast given in section section 2.3.3, and Atomic_Broadcast_Data is supposed to offer a totally ordered reliable multicast. Of course, more types could be included in the proposal for different FIFO and causal multicasts.

Any type derived from one of these three types would be a *coordinated type*: whenever the Write operation was invoked for an object of the type, it would be invoked on all partitions holding a replica of the object. Replicas could be created easily by declaring variables of the derived type in a normal package as these packages are replicated anyway. Yet all replicas would have the same type because the three types declared in fig. 4.1 all are remote types. The compiler would have to generate special RPC stubs for the primitive operations Write that would not call the standard interface to the PCS in System.RPC but an extension offering routines to broadcast RPC requests with the required semantics.

```
package Coordinated_Types is
  pragma Remote_Types;

  type Broadcast_Data is abstract tagged private;

  procedure Write (Data : Broadcast_Data)
    is abstract;

  type Reliable_Broadcast_Data is abstract tagged private;

  procedure Write (Data : Reliable_Broadcast_Data)
    is abstract;

  type Atomic_Broadcast_Data is abstract tagged private;

  procedure Write (Data : Atomic_Broadcast_Data)
    is abstract;

private
  -- Specified by the implementation
end Coordinated_Types;
```

**Fig. 4.1:** Specification of Coordinated Types

However, there are problems with this approach, too. As proposed, only the primitive operation `Write` would be broadcast to all replicas, opening wide the door to misuses of the feature. Consider the example in fig. 4.2 where a new type is derived from `Broadcast_Data`

```
with Controlled_Types;
use  Controlled_Types;
package Counters is

  type Counter_Type is new Broadcast_Data with
    record
      Value : Integer := 0;
    end record;

  procedure Write (Data : in     Counter_Type);

  procedure Read  (Data :    out Counter_Type);

  procedure Inc   (Data : in out Counter_Type);

  procedure Dec   (Data : in out Counter_Type);

end Counters;
```

**Fig. 4.2:** Deriving from a Coordinated Type

to implement replicated counters. Calls to `Read` will be ordinary procedure calls local to the partition, but so will calls to `Inc` and `Dec`! The implementation of procedure `Inc` must therefore be done in terms of `Write` instead of using the more natural "`Value := Value + 1`", and likewise for `Dec`. But there is nothing in the proposal that would forbid this second, wrong implementation.

On a far more basic level the question arises what exactly `Write` is supposed to do. It is a primitive operation of a coordinated type, so logically it should modify some variable of that type. But which variable? The one given as parameter cannot be modified as it is of mode `in`. Is the operation a misnomer and should not be called `Write` but rather `Update_All_Replicas`? But then, how can we find out which variable in the replicas should be changed? Consider also local variables of a coordinated type: how could an implementation of this scheme distinguish reliably between two different instances of a local variable in two different invocations of a subprogram? Again, the proposal makes no indication on how this case should be handled.

Indeed, the examples in [WB96] all use coordinated types only as an interface to some global state that in reality is implemented quite differently. The `Write` operation actually updates the state through side–effects; its controlling argument of the coordinated type serves only as a data container. This can be very misleading, and the dual implementation may cause serious maintenance problems, in particular if the state contains complex data structures.

Coordinated types cannot be used like other types in Ada 95; they are not suited for implementing abstract data types such as the `Counter` example above. They do not fit the common notion of a type, i.e., a definition of a set of values together with a set of operations that may be applied to them. Instead, they offer a fairly complex interface to a broadcasting facility, which is rather prone to being misused. I think coordinated types should only be extended in the (static) data dimension by adding new components to the record, depending on the needs of an application, but not in the operation dimension by adding new primitive operations for the derived types. (Following the spirit of type extension as it was defined for the original Oberon language [Wir88b]; alas, Ada 95 does not offer a way to restrict adding primitive operations[1].)

### 4.3.3 Packages

A normal package in Ada 95, i.e., one that doesn't contain a categorization pragma, is copied on each partition it is assigned to in the configuration process. In a certain sense, the package is replicated, but there is absolutely no coordination between these copies — the state of one particular instance of a normal package evolves independently from that of all other instances on other partitions. Furthermore, normal packages cannot be accessed remotely.

On the other hand, `Remote_Call_Interface` packages are remotely accessible, but they can be assigned to at most one single partition, i.e. their body — and therefore their state — exists only once in the application; it is not replicated.

A cross between these two kinds of packages might be just what is needed to express replication in a convenient manner in Ada 95. Wellings and Burns [WB96] also considered the introduction of a new categorization pragma `Replicated_Call_Interface` which would

1. I do not argue that Oberon was better in that respect: that language lacked extensibility in the operation dimension, i.e., *only* data extension was supported while new primitive operations could not be added. This flaw was later rectified in Oberon–2 [MW91].

have the same restrictions as the already existing pragma `Remote_Call_Interface` except for allowing the package to be assigned to several partitions. The semantics would be that each call to a subprogram declared in the visible part of the specification of such a package would be sent to *all* replicas of the package using some kind of broadcast. The proposal leaves open how the semantics of this broadcast is chosen; one could imagine an argument to the `Replicated_Call_Interface` pragma for this, or a pragma applying to a subprogram. The assumption with this approach is of course that either all replicas were deterministic or that any divergence of their states was not significant.

Although appealing at first sight, this approach has a number of shortcomings. First of all, one cannot guarantee determinism in an Ada 95 partition. It is therefore the programmer's responsibility to ensure the above constraint on determinism is met since the language cannot help in any way to verify it. This runs contrary to the spirit of Ada 95, which generally tries to assist the developer in avoiding errors (e.g., the rules for access types are such that dangling references cannot occur). Also, the approach implicitly assumes active replication; if it were to be used for passive replication, some additional requirements would have to be satisfied: the configuration language would have to provide some way to indicate a primary replica, and the code generated for such a package (or the underlying run–time support) would have to be able to function either as primary or as a backup. Furthermore, it is unclear what happens if the body of a replicated package uses other, normal packages. As those do exist on all partitions using them, but are not coordinated among themselves, ensuring replica determinism might become arbitrarily complicated in such a case.

On a more structural level the question arises what replication of single packages would be good for. Assigning packages to partitions normally is more than just cutting up the application in an arbitrary way and distributing the pieces: a partition usually groups packages that logically belong together. Consider now what happens on a node failure. Unless the replicated package (with pragma `Replicated_Call_Interface`) is the only RCI package in a partition, the application will lack the functionalities offered by other packages in the partition on the failed node. Therefore, it is highly likely that the RCI packages in a partition would be either all `Replicated_Call_Interface` packages (if replication is desired), or all `Remote_Call_Interface` packages (if replication is not required). Mixtures of these two kinds of packages do not seem meaningful. This is a strong hint that packages are the wrong level of abstraction for replication, and that the unit of replication should indeed be the same as the unit of distribution: the partition.

### 4.3.4 Partitions

For full fault tolerance, the partition is the appropriate choice as replication unit. It corresponds to the unit of distribution (virtual node) in Ada 95, and as such is also the most logical choice.

Note that several partitions may be grouped together on the same physical node. This allows an argument similar to the one made above for replicated packages: it is probable that either all or no partitions on a given node will be replicated, and therefore the node should be the unit of replication. While this may be true, “physical nodes” are a concept beyond what the language standard covers, and using this even larger granularity would lead to a loss in functionality. If one really wanted to have a mixture of replicated and non–replicated packages on a node, one can still achieve this effect by appropriately assigning the packages to partitions and then allocating several partitions on one node. This would be impossible if the unit of replication were the node.

## 4.4 Non–Determinism in Ada 95

With the choice of the partition as the basic replication unit, the “replicated objects” the title of this thesis refers to are the partitions of a distributed Ada 95 application. Partitions in Ada 95 generally execute in a non–deterministic way. In this section, I explore the causes for this non–deterministic behavior and give an introductory discussion of some basic attempts to handle or avoid this.

### 4.4.1 Causes of Non–Determinism in Ada 95

The most obvious causes of non–determinism in the language are all related to tasking:

- Task scheduling may be non–deterministic: if there are several tasks running in a partition, their interleaving and sequence of execution is not specified. In particular, pre–emptive tasking implementations may cause problems in this respect.
- The language standard does not impose any order on the choice if more than one alternative of a selective accept is open. The same holds also for entries of protected objects. ([ISO95, 9.5.3(17)]).
- As a special case of this, an arbitrary choice is made between multiple `delay` alternatives of a selective accept that have all expired [ISO95, 9.7.1(18)].
- It is not defined when exactly the abortable part of an asynchronous `select` statement is aborted when the triggering statement completes.

Explicit dependencies on time also are a source of non–determinism, in particular when used in the context of an asynchronous transfer of control or a timed entry call. Because there is no global notion of time 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. The same reason may also cause a `delay` alternative of a selective accept to be open at a given moment on one replica while it is still closed, i.e., has not yet expired, on another replica.

The tasking support within the run–time support will follow the same rules on all replicas. Since all replicas are copies of the same partition — and *are* the same partition for the rest of the distributed Ada 95 application — they all use the same task dispatching, locking, and queueing policies. Execution may be non–deterministic in certain cases nonetheless. Implicit timing dependencies, for instance the use of time–sliced pre–emptive task scheduling, may cause executions on different replicas to diverge; and if replicas execute on true multi–processors, all bets are off unless physical nodes are homogeneous and synchronized hardware–wise down to the level of the hardware clock.

### 4.4.2 Is Enforcing Deterministic Behavior Possible?

One approach to avoiding this non–determinism is the use of special pragmas. The language standard offers in its real time systems annex [ISO95, annex D] a couple of pragmas to choose a particular task scheduling policy, locking policy for protected objects, or queueing policy for entry calls. Using these or additional, new pragmas one could basically force tasking to behave in a deterministic way, i.e. in such a way that the same task scheduling decisions are made at each task dispatching point given the same task set. However, these restrictions to achieve determinism are severe and greatly reduce the field of applicability of the approach, and furthermore, it is far from easy to define a deterministic tasking scheme. In fact, the guidelines for development of safety critical systems often forbid the use of tasks altogether; recently, attempts to define a restricted tasking model for such systems have been undertaken [BDR98], but the set of restrictions is impressive, tailored specifically to the field of safety critical real–time systems, and far from transparent.

One could try to solve the problems related to time by disallowing the use of explicit timing dependencies. The safety and security annex of the language standard [ISO95, annex H] offers a set of restrictions that may be used in conjunction with pragma `Restrictions`. One of these restrictions is `No_Delay` and forbids the use of `delay` statements and semantic dependence on package `Ada.Calendar`, but allows the use of `Ada.Real_Time`. Although the latter is supposed to represent physical time as observed in the external environment (the so–called “wall clock time”), there is no guarantee that the time bases of two physical machines executing two replicas of a partition are synchronized. Therefore, even this restriction is not sufficient to *guarantee* the deterministic behavior of a partition. Furthermore, forbidding the use of `delay` statements is not transparent and might make it impossible to use third–party libraries, which might contain delays without the application (or the developer, respectively) using them ever becoming aware of this fact.

Also note that restrictions generally apply to the whole partition [ISO95, 13.12(8)], not just to the library unit containing the pragma. This includes the run–time support, and in particular the partition communication subsystem! Delays are frequently used in communication protocols, and the exclusion of the `delay` statement would make the implementation of a PCS more difficult. (The PCS of GNAT, GLADE, does use delays.)

Even if this problem were solved, e.g., through a new restriction that would apply only to the application, but not to the run–time support itself, simply choosing deterministic tasking policies and forbidding explicit dependencies on time would not suffice to ensure replica determinism. Consider the simple server shown in fig. 4.3 below.

```
package Example is
  pragma Remote_Call_Interface;

  procedure Do_It;

  procedure Reset;

end Example;
```

```
package body Example is

  protected State is
    procedure Set (I : Integer);
    function  Get return Integer;
    procedure Inc;
  private
    X : Integer := 0;
  end State;

  protected body State is

    procedure Set (I : Integer) is
    begin
      X := I;
    end Set;

    function Get return Integer is
    begin
      return X;
    end Get;

    procedure Inc is
    begin
      X := X + 1;
    end Inc;

  end State;

  procedure Do_It is
  begin
    --  Calculate something for
    --  some time, then do:
    State.Inc;
  end Do_It;

  procedure Reset is
  begin
    State.Set (42);
  end Reset;

end Example;
```

**Fig. 4.3:** A Simple Server Example

Assuming that this package is the only package of the partition, the server in fig. 4.3 offers the two operations `Do_It` and `Reset`. If it is replicated using the determinism restrictions sketched above, it is still possible that two replicas evolve differently because the creation of the tasks executing the RPCs is triggered by external events: the arrival of the RPC request messages. Even if replicas are organized such that they all handle all requests in the same order, this cannot guarantee that the time between the handling of two requests is exactly the same on all replicas. It is therefore possible to get different schedules as shown in fig. 4.4.

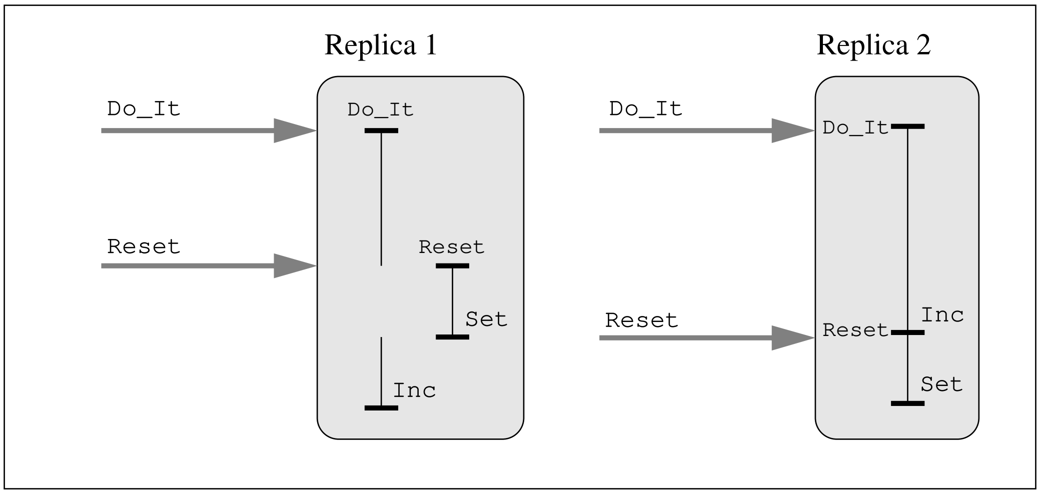


**Fig. 4.4:** Diverging States in a Multithreaded Partition

Task scheduling — even if it is deterministic, i.e., makes the same decisions given the same set of tasks — might lead to a situation where `State.Set (42)` from the `Reset` remote call is executed before the `State.Inc` from the `Do_It` procedure on one replica (depicted on the left side in fig. 4.4), while the other replica executes the two state changes in the opposite order, simply because the elapsed time between the two requests is larger on the second replica. Although both replicas treat the two external *requests* `Do_It` and `Reset` in the same order, the sets of active tasks on the two replicas is not the same at all times. On replica 2, the deterministic tasking support takes a different decision since the task executing `Reset` doesn't exist yet because the RPC request message didn't arrive yet. As a result, the states of the replicas diverge because they do not execute the *state changes* in the same order!

There is no way to synchronize the arrival and handling of RPC requests exactly since there is no common time base between different replicas. One way to avoid this problem would be to forbid concurrent execution of RPC requests altogether, but such an approach is not feasible since it would result in a global serialization of requests throughout the distributed application.

The conclusion is that replicated multithreaded partitions inherently are non–deterministic, even if all the non–deterministic aspects of the language are eliminated. Even if a deterministic task dispatching policy is used to avoid implicit timing dependencies and all explicit timing dependencies are avoided, there still remains an implicit timing dependency that cannot be removed. There is no way to enforce determinism on the tasking level alone — it is always necessary to *enforce a common sequence of state changes*.

### 4.4.3 Active Replication Using Consensus

Active replication is attractive because it offers a high availability of the replicated object: as replicas execute in parallel, failures do not incur an additional overhead. However, simple active replication without some sort of explicit synchronization between the replicas is not possible in Ada 95 as I have shown in section 4.4.2.

The non–determinism in fig. 4.4 stems from the fact that message deliveries are not synchronized with task scheduling — basically, the non–determinism in the communication layer is not accounted for. It seems that this could be remedied for active replication by forcing the replicas to reach consensus (cf. section 2.3.3) on all task scheduling decisions, *including* message deliveries.

On first sight, this approach seems feasible if one assumes deterministic task scheduling. The basic idea is as follows. Replicas are actively replicated and execute all the same code. Therefore they all start in the same state with the same sets of tasks. Because task scheduling is deterministic, they will all make the same scheduling decisions in the same order and their states will therefore evolve coherently. The only possibility for divergence arises from message deliveries. Replicas are forced to reach a consensus as to *when* to deliver a message, such that they all deliver it in the same logical moment.

With deterministic tasking, progress can be measured by the task dispatching points [ISO95, D.2.1(4)] a task has passed. (See also the discussion of events in chapter 6.) Consensus for message deliveries must then be reached on the task dispatching point at which a message is delivered[1]. The most advanced task dispatching point must be taken as the common consensus value[2]. 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.

In the example in fig. 4.4, replica 2 might already have completed the remote call `Do_It` when replica 1 starts the consensus. As it has advanced further than replica 1, this will be the consensus value, and consequently, replica 1 will also delay the delivery of the RPC request message for `Reset` until `Do_It` has completed. Thus they will both interleave `Do_It` and `Reset` in the same way (or, in this example, execute one after the other), and thus replica consistency is ensured.

The overhead of an additional consensus for each message delivery may be significant, but seems still tolerable. However, this scheme is *not* transparent: it can only work if all timing dependencies are disallowed. This includes calls to `Ada.Calendar` or

1. Besides an earlier consensus needed for the totally ordered multicast necessary to ensure the ordering condition given in section 2.3.2.
2. Although all replicas make the same scheduling decisions, they do not execute in lockstep: one replica may already have advanced further than another.

`Ada.Real_Time`; and of course `delay` statements must not be used in a replica, otherwise replicas can diverge because their clocks are not synchronized.

If these timing dependencies are to be allowed in the interest of transparency, the replicas must reach consensus on *each and every* task scheduling decision, 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.

This is clearly impractical as it implies that replicas communicate over the network at all task dispatching points to reach consensus. Forcing the replicas to execute an interactive agreement protocol for each task scheduling decision would tremendously slow down the partition[1]. In fact, one loses the main advantage of active replication, which is its high availability.

The idea of synchronizing message deliveries and task scheduling is therefore not suitable for active replication of general partitions. Yet it should not be dismissed altogether: in chapter 6, I apply a very similar idea in the context of semi–active replication. There, the synchronization overhead can be bounded precisely because the replicas do *not* have to communicate with each other for each and every scheduling decision.

In the following chapter, I consider a different approach to make all replicas follow the same sequence of state changes. It tries to avoid synchronizing task scheduling altogether by identifying state accesses and then synchronizing only those, using a coordinator–cohort replication scheme.

1. The overhead may be fairly high even in dedicated, special–purpose systems where tasking is restricted and where consensus only must be reached at certain points. For instance, the SIFT [WLG+78] and FTMP [HSL78] architectures for reliable hard real–time systems (both synchronous systems using very restricted tasking models) incur synchronization overheads of 60 to 80 percent [Pra96, p. 272]. A more recent system, MAFT [KWFT88], which also uses a restricted tasking model, achieves better performance primarily by delegating agreement to special hardware. The MARS project [KFG+93] also employs active replication, but again, tasking is restricted (to static scheduling even).

# Chapter 5:
# Non–Deterministic Replicas

In this chapter, I consider an approach to ensure identical evolution of all replicas' states by synchronizing state accesses. State accesses are identified and the execution of the replicas is coordinated such that they all perform the accesses in the same sequence. Otherwise, the execution of partitions is assumed to be non–deterministic: the tasking system is viewed as a black box.

## 5.1 The Computation Model

In this approach, I assume the non–deterministic computation model defined for Ada 95 by the language standard [ISO95]. Partitions that are to tolerate crash failures of physical nodes are replicated on several physical nodes; all the replicas of a partition constitute a group.

For the purposes of discussion in this chapter, I assume a very restricted model. I only consider S– and CS–partitions that are basically inactive; all activity is triggered by the reception of RPC requests. As mentioned in section 3.2.5, a partition handles remote calls concurrently: each remotely called subprogram is executed by a task. I assume that these tasks communicate with each other only through protected objects. This model of course is far more restrictive than what Ada 95 in general allows, but for the purpose of this discussion, it is already sufficiently complicated.

The choice of this computation model has far–reaching consequences. Active replication is clearly not possible in this context. I will therefore consider a passive replication scheme, i.e., coordinator–cohort replication. Furthermore, it is not possible to take intermediary checkpoints while an RPC is in progress: such a checkpoint would have to include system state (such as the currently active task, even the current value of the PC (program counter) and of the other registers of a physical node, the state of the stack, etc.) that is not accessible and that might be meaningless for another replica in a heterogeneous distributed system.

## 5.2 Coordinator–Cohort Replication

The coordinator–cohort replication scheme was first introduced in the context of the Isis system [BJRA85, Bir85]. Replicas are organized as a group, and requests are multicast to all group members. One of the replicas is designated the coordinator, the others are its cohorts. The coordinator is the only replica that executes a request; periodically, it informs its cohorts on its progress by sending them checkpoints. If the coordinator fails while servicing a request, one of the cohorts is appointed the new coordinator and resumes processing of the request from the last checkpoint.

Checkpoints in Isis not only include the state of the data at a replica, but also the current execution state of a request: stack, program counter, and so on. If a failed coordinator had made nested calls to other groups since its last checkpoint, the new coordinator will redo these calls. The receiver detects these duplicate messages and returns the retained results from the original invocation [BJRA85]. Note that this implies that requests have deterministic behavior. Also note that in the Isis approach, intermediary checkpoints are not necessary, it would suffice that the coordinator inform the cohorts of state changes after a request has terminated. If a failure occurred, the cohort would simply restart the request from the beginning.

Requests can be handled concurrently, and the coordinator may be chosen on a per–request basis in Isis: for each request, a different replica may be chosen as coordinator. As I show in section 5.3 below, concurrent handling of requests requires that requests be implemented as nested transactions.

## 5.3 Analysis

RPCs in Ada 95 (both synchronous and asynchronous ones) are defined to execute with "at most once" semantics: if the called subprogram returns normally, its body has been executed exactly once as the result of a call [ISO95, E.4(11), E.4.1(10)]. If the predefined exception `Communication_Error` is raised on the calling partition, the remotely called procedure may not have been executed.

The consequences of "at most once" semantics significantly complicate matters when considering CS–components, i.e., a partition that is a server and a client at the same time. Consider the scenario shown in fig. 5.1a: CS is a replicated (duplicated) partition with replicas $R_1$ and $R_2$ using coordinator–cohort replication. Some partition makes a remote call on this replicated partition (RPC request $\mathtt{req_A}$), which is multicast to the group.

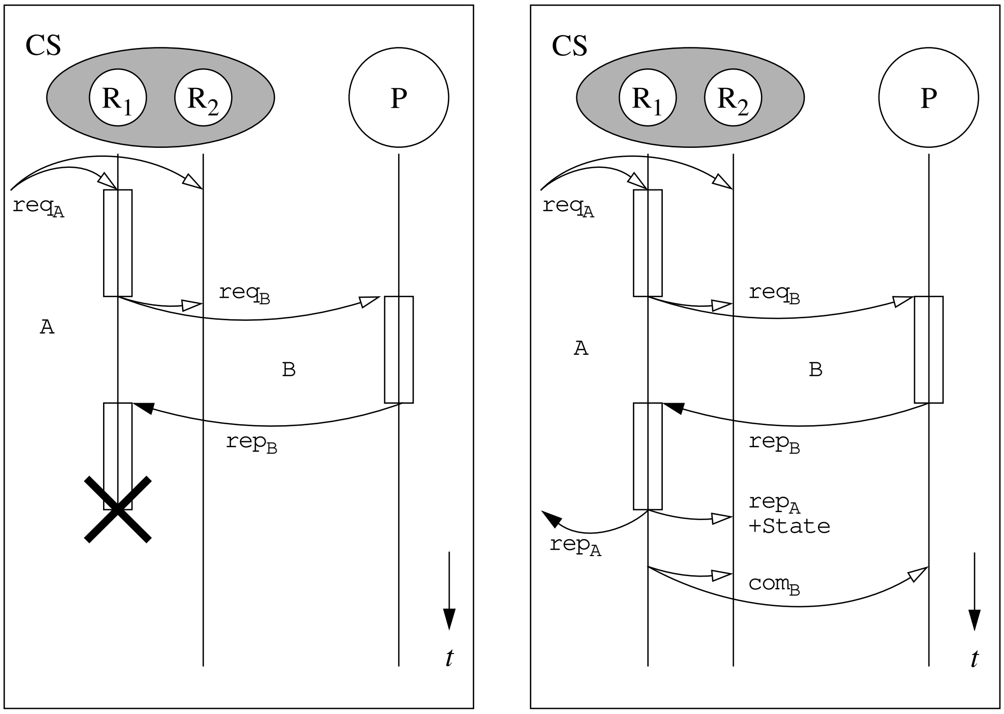


**Fig. 5.1a:** Nested RPC **Fig. 5.1b:** Committing Nested Calls

The coordinator ($R_1$) then begins executing the remote call, but to fulfill the request, it has to make a nested remote call $\mathtt{req_B}$ to another partition P, which may be replicated or not. This nested call is successful, but some time after, the coordinator $R_1$ fails, and the cohort $R_2$ becomes the new coordinator and takes over. $R_2$ “knows” there is an outstanding RPC request $\mathtt{req_A}$ and that the former coordinator didn’t yet send back the reply $\mathtt{rep_A}$ (because it didn’t receive a checkpoint yet) and thus commences executing the subprogram again. However, due to the non–deterministic computation model, there is no guarantee whatsoever that the path of execution on $R_2$ is the same as the one that had been taken on $R_1$! It is not even sure that $R_2$ will redo the nested remote call to P, or if it does, that it will pass the same parameters.

To satisfy the “at most once” requirement and to maintain the coherence of the overall distributed state, the nested remote call to P must be undone — partition P must *roll back* to the state before `B` was executed. Note that it cannot rely on a technique to recognize duplicate invocations, discard the second one and just return the results (if any) from the first one: it is not sure there will be a second invocation, in which case the overall behavior of the application appears to an omniscient observer as if P executed the remote call `B` spontaneously, which violates the “at most once” rule.

Consider now the case where $R_1$ does not fail (fig. 5.1b). Request $\texttt{req}_\texttt{A}$ terminates successfully, and a reply $\texttt{rep}_\texttt{A}$ is sent back to the caller. In order to be able to roll back B, partition P will have maintained some data structures. These can now be released, and CS therefore informs P that B may now be considered permanent by sending a commitment $\texttt{com}_\texttt{B}$. Of course, I assume that the RPC A itself is permanent at that instant — either because $\texttt{req}_\texttt{A}$ is not a nested RPC request (i.e., originated on a C–component), or, if it is, because CS received a commitment notification for it.

Undoing nested RPCs if the caller fails can have undesirable side effects if a partition handles incoming RPC requests concurrently, as it is suggested in the Ada 95 language standard [ISO95, E.5(24, 25, 28)].

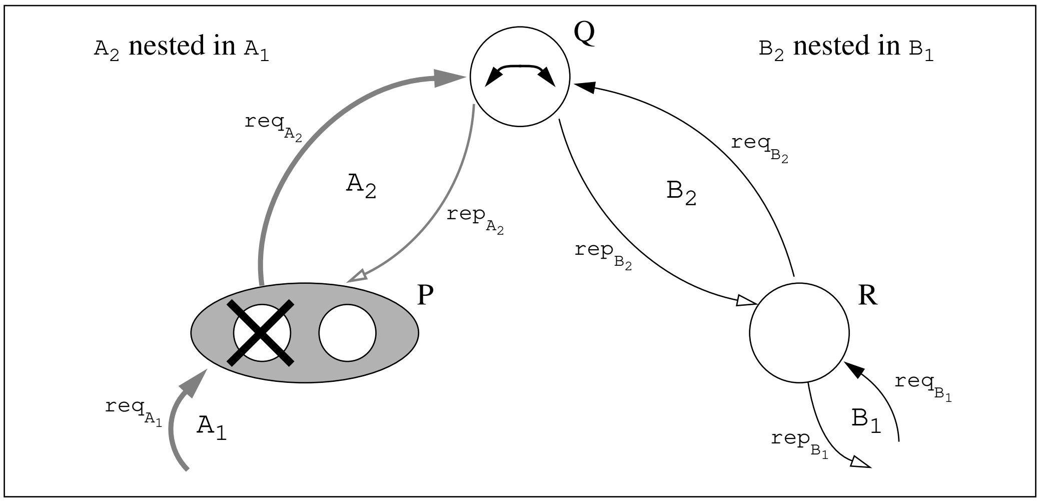


**Fig. 5.2:** Concurrent RPCs + Rollback = Domino Effect

Consider the case in fig. 5.2, where partition Q, which may be replicated or not, handles the two nested remote calls $A_2$ and $B_2$ concurrently. Partition P is replicated, and, as in the example above, its coordinator fails after the nested call $A_2$ has completed, but before the "parent" remote call $A_1$ has terminated. As shown above, this requires that the nested remote call $A_2$ be undone.

The problem in this scenario is that $A_2$ may have communicated with the concurrently running remote call $B_2$ originating on partition R, e.g. through a protected object located on partition Q where $A_2$ might have written a value that $B_2$ subsequently read. If Q rolls back to a state before the execution of $A_2$, it will also have to undo $B_2$, and therefore even partition R must undo $B_1$ even if this remote call has already terminated successfully. The combination of concurrent execution of remote calls and the requirement to roll back results in a potentially disastrous *domino effect* where rollbacks proliferate throughout the distributed application even if only one replica fails.

Remote calls under a non–deterministic computation model must therefore be synchronized to avoid the domino effect. A concurrency control scheme must govern the execution of concurrent remote calls in such a way that a domino effect cannot occur:

concurrent remote calls must execute isolated from each other; the state changes a remote call makes may become visible to other remote calls only once it has been committed. RPCs must have *transactional semantics* in the non–deterministic computation model.

## 5.4 Transactions

Transactions are a classic software structure for managing concurrent accesses to global data and for maintaining data consistency in the presence of failures. A transaction is a software construct satisfying the ACID properties: (failure) *atomicity*, *consistency*, *isolation*, and *durability* (see section 2.4.4).

The classic definition of transactions based on these four conditions is very much database–oriented. For the purpose of fault tolerance in the context of a general–purpose programming language, consistency and durability are problematic requirements. I view consistency generally rather as a responsibility of the developer, who has to ensure that his application properly maintains its data; the only consistency guarantee a transaction can give is that its execution does not *erroneously* corrupt the system state. Durability also has to be qualified: whether the updates have to be written to stable storage again depends on the application's specification. In the context of fault tolerance by replication, durability shall denote the fact that all replicas are aware of the updates, which therefore can survive a crash failure of a physical node.

### 5.4.1 Serializability

*Isolation* of transactions is ensured by a *concurrency control* scheme governing the execution of concurrent transactions. It schedules the operations of multiple transactions in such a way that the consistency criterion of *serializability* [Pap79] is met. I define serializability following the notation of Weihl [Wei89][1], using the concept of histories introduced in section 2.3.1 on page 15, with the addition that $\mathbf{H}|_t$ denotes the subhistory of transaction $t$ and that the system is composed of a set of transactions $\mathbf{T} = \{t_1, ..., t_n\}$ and a set of objects $\mathbf{O} = \{x_1, ..., x_m\}$. Furthermore, let the concatenation of two histories be denoted by the operator $\bullet$. A history $\mathbf{H}$ is *serial* if the operations of different transactions are not interleaved:

*Definition 5.1: Serial and Serializable Histories*

- A history $\mathbf{H}$ is *serial in some order* $<_T$ on $\mathbf{T}$, if $\mathbf{H} = \mathbf{H}_{<_\mathrm{T}} = \mathbf{H}|_{t_{i_1}} \bullet ... \bullet \mathbf{H}|_{t_{i_n}}$, where $i_p < i_q$ if $t_p <_\mathrm{T} t_q$ for all $p, q \in [1 .. \mathrm{n}], p \neq q$.
- A history $\mathbf{H}$ is *serializable in some order* $<_T$ if there exists a legal equivalent serial history $\mathbf{H}'_{<_\mathrm{T}}$.

1. For another, perhaps more classical formulation, see e.g. [Wei88].

Classical database theory defines serializability with respect to the "reads" relation [BHG87]. Operations $o_i$ of transactions are restricted to read $r(x)$ and write $w(x)$ operations on objects $x$. The "reads" relation $<_R$ for a history **H** is defined by

*Definition 5.2: The "reads" relation* $<_R$

- $o_i <_R o_j$ if $o_i < o_j \wedge$
  $(\exists x \in \mathbf{O}: o_i = w(x) \wedge o_j = r(x) \wedge \neg(\exists o \in \mathbf{H}: o = w(x) \wedge (o_i < o \wedge o < o_j)))$.

The last term of this conjuction simply expresses that no other writes on $x$ in the history happen between $o_i$ and $o_j$. Transactional serializability requires that executions must be serializable in the order $<_R$:

*Definition 5.3: Serializability*

- A history **H** is *serializable* if there exists an equivalent serial history $\mathbf{H}'_{<_R}$.

In practice, transactional systems often implement *conflict serializability*, as it is simpler and implies serializability. It is based on the notion of the conflict dependence relation:

*Definition 5.4: Conflict Dependence*

- Two operations $o_i$ and $o_j$ *conflict* if they both operate on the same object $x$ and at least one of them is a write operation $w(x)$: $(o_i = w(x) \wedge (o_j = r(x) \vee o_j = w(x))) \vee (o_j = w(x) \wedge (o_i = r(x) \vee o_i = w(x)))$.
- $o_j$ *depends on* $o_i$, written as $o_i <_D o_j$, if $o_i$ and $o_j$ conflict and $o_i < o_j$.

*Definition 5.5: Conflict Serializability*

- A history **H** is *conflict serializable* if there exists an equivalent serial history $\mathbf{H}'_{<_D}$.

Note that the definitions of serializability and conflict serializability do not depend on the sequential specifications of the objects anymore (the equivalent serial history **H'** is not required to be "legal"). This is because the relations $<_R$ and $<_D$ already encode legality conditions for objects that can only be written or read; the type of the objects therefore becomes unimportant. For this reason, classical databases are sometimes called *untyped* [BL93].

### 5.4.2 Concurrency Control

There are many concurrency control protocols described in the literature. Basically one distinguishes pessimistic protocols (locking or timestamp–based) and optimistic protocols. Optimistic protocols allow concurrent transactions to proceed until they want to commit, at which time the concurrency control component verifies that there are no conflicts, and if there are, aborts (some of) the offending transactions. I will only consider pessimistic protocols.

A classic concurrency control scheme that ensures serializability is *two–phase locking* (2PL) [Pap79]. This protocol is defined by the two rules below:

*Definition 5.6: Two–Phase Locking Rules*

- Transactions lock the objects they access such that conflicting operations of other transactions are blocked until the lock is released.
- Once a transaction has released a lock, it is not allowed to acquire other locks.

In order to avoid domino effects, 2PL is extended to *strict 2PL* by the following additional condition:

*Definition 5.7: Strict 2PL Rule*

- A transaction holds all acquired locks until EOT.

*Strict* protocols avoid domino effects (or *cascading aborts*, as they are sometimes called in the database world) because they make sure that no transaction can read a state modification before the transaction that made it has committed. Subsequently, "2PL" always denotes strict two–phase locking.

The conflict relation above is defined for read and write accesses, so it is only natural to introduce read and write locks on objects. 2PL for read/write locking follows the usual rules, i.e., multiple readers of an object are allowed, but only one writer:

- A transaction may acquire a *read* lock on an object only if no other transaction holds a write lock on it.
- A transaction is granted a *write* lock on an object only if no other transaction holds any lock on the object.

With locking schemes, deadlocks can occur, i.e. situations where two transactions wait for locks on objects held by the other transaction. There are three ways to deal with this problem: deadlock *prevention* (scheduling transactions in such a way that deadlocks cannot occur in the first place; this is only feasible in systems with a fairly rigid pre–defined structure), deadlock *avoidance* such as the wound–wait and wait–die algorithms that avoid deadlocks at the cost of perhaps unnecessary abortions of transactions if they request a lock that might lead to deadlock, and finally deadlock *detection*.

Deadlock detection examines the relations of transactions waiting to be granted locks (the *waits–for* graph) for cycles (i.e., deadlocks) and resolves the deadlock by aborting a transaction in the cycle. In a distributed setting, this is somewhat complicated by the fact that no node has a complete view of the waits–for graph.

A completely different class of transaction scheduling algorithms ensuring serializability are timestamp–based concurrency protocols. These assign each transaction $t$ a unique timestamp $t.ts$ at BOT and maintain two timestamps for each object $x$: $x.wt$ records the largest $t.ts$ of all $t$ that wrote the object, $x.rt$ records the largest $t.ts$ of any $t$ that read it.

The protocol guarantees that any conflicting operations are executed in timestamp order, aborting and restarting transactions if necessary. The basic rules are as follows:

*Definition 5.8: Timestamp Ordering Rules*

- If transaction $t$ wants to perform a read operation $r(x)$ on an object $x$, the scheduler allows the read only if $t.ts \geq x.wt$ and sets $x.rt := max\ (x.rt,\ t.ts)$. If $t.ts < x.wt$, the value that should be read has already been overwritten, and $t$ is aborted and restarted.
- If $t$ wants to do a write access $w(x)$, two checks are made: if $t.ts < x.rt$, or $t.ts < x.wt$, $t$ is too late and is aborted and restarted; otherwise, $w(x)$ is executed and $x.wt := t.ts$.

When a transaction is restarted, it gets a new timestamp in order to avoid certain abortion. This is different from the aforementioned wound–wait and wait–die deadlock avoidance systems (which also are timestamp–based), where a transaction keeps its original timestamp when it is restarted. The above two rules ensure serializability, but still allow cascading aborts to occur. *Strict* timestamp–ordering is a modification to avoid cascading aborts. The basic idea is to delay all conflicting operations of transactions $t_j$ with $t_j.ts \geq t_i.ts$ on an object $x$ once it has been modified by some transaction $t_i$ until $t_i$ commits[1]; see [BHG87] for the details.

Timestamp ordering and 2PL are complementary concurrency control schemes — both allow some execution histories the other doesn't allow. It is not clear which one would be "better" in the context of a general–purpose programming language such as Ada 95. On the one hand, two–phase locking may deadlock and require deadlock resolution; on the other hand, timestamp ordering is deadlock–free (as no waiting occurs), but may cause unnecessary aborts as in fig. 5.3.

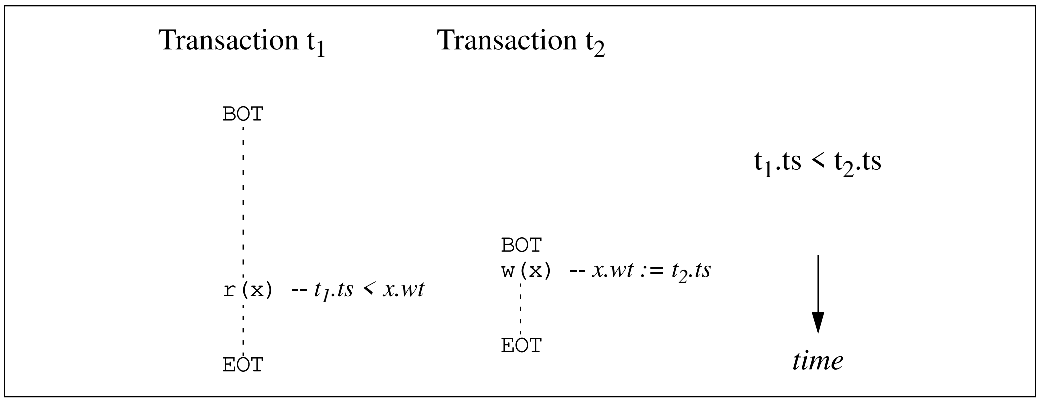


**Fig. 5.3:** Unnecessary Abort with Timestamp Ordering

In this example, the long–running transaction $t_1$, which has a smaller timestamp than $t_2$, is aborted and restarted because the object $x$ has already been written by $t_2$ by the time $t_1$ tries to read it. Transaction $t_1$ is therefore considered to be "too late" and is aborted,

1. The resulting strict protocol is still deadlock–free as waiting occurs only if $t_j.ts \geq t_i.ts$.

although waiting for $t_2$ to commit before performing *r(x)* would have been a perfectly acceptable solution in this case. (Note that $t_1$ is aborted in both basic and strict timestamp ordering because its timestamp is too small.)

In *multi–version* concurrency control schemes, each write operation *w(x)* produces a new version of *x*, so writes can never conflict. Read operations *r(x)* not necessarily return the "latest" version of *x*; the scheduler is free to make them read from older versions. The benefit is that late read operations do not cause the issuing transaction to be aborted. The correctness criterion is extended to *one–copy serializability*, i.e., a multi–version history $\mathbf{H}_\mathrm{M}$ is one–copy serializable if there exists an equivalent one–copy serial history $\mathbf{H}_{<_R}$ such that if $t_i <_\mathrm{R} t_j$, then $t_i$ is the last transaction preceding $t_j$ that wrote to *any* version of the objects involved in the "reads from" relation. Both 2PL and timestamp ordering can be used for multi–version concurrency control. For an extended discussion of these topics, see [BHG87].

### 5.4.3 Recovery

The (failure) *atomicity* property of transactions is implemented by a recovery manager. It guarantees that the state contains only the effects of committed transactions in case of a failure (or a voluntary abort of a transaction).

Recovery techniques for databases usually distinguish three different classes of recovery: *abort* recovery, which is concerned with undoing effects of a single transaction when an abort occurs; *crash* recovery dealing with restoring the database in non–volatile memory to a consistent state despite the loss of data in volatile memory (buffers etc.); and *archive* recovery, which restores the database after a media failure such as a head crash on the harddisk.

For abort recovery, there exist two basic approaches [HR83]:

- *Update–in–place*, where write accesses *w(x)* are performed immediately and the old values of the objects (their *before–images*) are retained in an *undo log*. If a transaction commits, its undo log is simply discarded; if it aborts, the undo log is used to restore the original object states.
- *Deferred update*, where write accesses *w(x)* append entries to a *redo log* (also called an *intentions list*). Subsequent reads get the latest value from the redo log (*after–images*). If a transaction commits, the changes recorded in the redo log are applied to the state; if it aborts, the redo log is discarded.

Crash recovery uses logs to implement failure atomicity. Transactions write their undo and redo entries in this log; they also write abort and commit entries to it. The log is forced onto stable storage when a transaction commits, and before state updates (*write–ahead logging*). This log on stable storage is used after a node crash incurring loss of the volatile memory to reconstruct the state: transactions for which the log contains a commit entry but whose effects have not yet been reflected in the state are redone, those for which no commit entry exists are undone. Recovery in classical transactional databases in discussed in numerous publications, see e.g. [BHG87, MHL+92, Wei93, Mad98].

In distributed systems, additional complexity is caused by the requirement that all the virtual nodes that participate in a transaction reach agreement on whether to commit or abort the transaction. The most common solution to this *atomic commitment* problem is the *two–phase commit* protocol [Gra78, ML83]. In this protocol, one participant acts as a coordinator. The protocol has two phases: in the preparation phase, the coordinator collects each node's opinion on whether to abort or commit; in a second phase, the coordinator then sends the decision to all participants. This protocol is blocking: if the coordinator fails at an inopportune moment, participants may remain blocked waiting for the decision to arrive. The *three–phase commit* solves this problem for reliable networks, but incurs a vastly higher performance penalty and is much more complicated. Another approach is to note the close similarity of atomic commitment to the consensus problem, and in fact, this road has recently led to non–blocking atomic commitment protocols that are simpler than three–phase commit ([BT93, GLS95]).

### 5.4.4 Nested Transactions

*Nested transactions* [Ree78, Mos81] extend transactions to allow nesting, which makes the whole concept interesting for general–purpose programming languages and also for distributed systems. Standard transactions are a sequence of simple operations that access the state. With nested transactions, an operation of a transaction itself may be another *subtransaction*, which in turn may have still further subtransactions. This gives rise to a tree of transactions, whose root is called a *top–level transaction*. Subtransactions that have no further child transactions are called *leaf* transactions; leaves need not be at the same level in the transaction tree.

Many models for nested transactions have been proposed; I will stick with Moss's original definition:

*Definition 5.9: Nested Transactions*

- A subtransaction executes concurrently and atomically with respect to its parent transaction and to all its sibling subtransactions.
- A subtransaction can commit or abort independently of its siblings or its parent.
- A subtransaction is isolated from its siblings and its parent.
- State accesses are done only in leaf transactions.[1]

1. This is a purely conceptual simplification. If the parent of a subtransaction should access the state, this can always be modeled by encapsulating these accesses in yet other (leaf) subtransactions. Implementations can "optimize away" these additional subtransactions.

If a (sub)transaction aborts, all its children (even if they already have committed) are aborted, too, its state changes are undone and its parent is notified, which may then take appropriate actions, e.g., retrying the aborted subtransaction, or trying to accomplish the work to be done by some other means, or aborting itself. A commit of a subtransaction is only conditional: if the parent transaction is later aborted, the subtransaction's state changes will still be undone. Only the commitment of top–level transactions is definitive. When a subtransaction commits, its state changes become visible only within its parent's subtree — other transactions (and their children) cannot "see" them, as transactions execute atomically and isolated from each other.

The serialization correctness criterion also applies to nested transactions [RA94]. From a parent transaction's point of view, a subtransaction is just another (indivisible) operation taking place between any two other operations. Hence, if the subtransaction is executed serially between these two operations, the overall behavior will be correct. Thus the requirements for concurrency control become:

*Definition 5.10: Serializability for Nested Transactions*

- The history **H** must be serializable with respect to top–level transactions.
- The subhistories $\mathbf{H}|_t$ of all parent transactions $t$ must be serializable with respect to $t$'s children.

Two–phase locking can be adapted to nested transactions using lock inheritance. Whenever a subtransaction aborts, its state changes are undone and the locks it is holding are released; when it commits, all the locks it is holding are passed on to its parent. The rules for 2PL read/write locking for nested transactions are as follows [Mos85]:

*Definition 5.11: 2PL for Nested Transactions*

- A subtransaction $t$ is granted a *write* lock on an object only if all other transactions holding a lock on that object are ancestors[1] of $t$.
- A subtransaction $t$ is granted a *read* lock only if all holders of write locks on that object are ancestors of $t$. (Note that $t$'s superiors — by definition — do not access the state concurrently with $t$, so no conflict arises.)
- When a subtransaction $t$ aborts, all its locks are discarded. If any of its superiors hold locks on the same objects as $t$ does, they continue to do so.
- Upon commit of a subtransaction $t$, the locks are inherited by $t$'s parent.

These rules ensure serializability for nested transaction histories; a proof is given in [Lyn83].

1. I follow Moss's terminology here: *ancestor* is the reflexive ancestor relation in the transaction tree, i.e., a (sub)transaction is an ancestor of itself; *superior* denotes the non–reflexive ancestor relation.

Timestamp ordering may also be used for serializing nested transactions: [Ree78] used a multi–version timestamp ordering scheme for concurrency control.

Abort recovery in nested transactions can be seen as maintaining a stack of versions for each object accessed by the subtransactions. Whenever a transaction accesses an object for the first time, it pushes its current value on the stack and subsequently modifies the version at the top of the stack. When a subtransaction commits, its version replaces that of its parent (if any; i.e., if the parent had its own version of the object, one pops twice and then pushes the child's version again; if the parent hadn't accessed the object yet, nothing has to be done); when it aborts, it simply pops its version from the stack. This stack concept can be implemented directly (giving a kind of deferred–update algorithm) as in Argus [LS83], or using update–in–place and keeping the before images in an undo log. Note that because a subtransaction only commits tentatively, the commit decision can be taken locally. Only the commitment of a top–level transaction requires using a full–fledged atomic commitment protocol.

## 5.5 An Approach in Ada 95

The basic idea of this approach to allow replication of non–deterministic partitions is sketched in fig. 5.4 below.

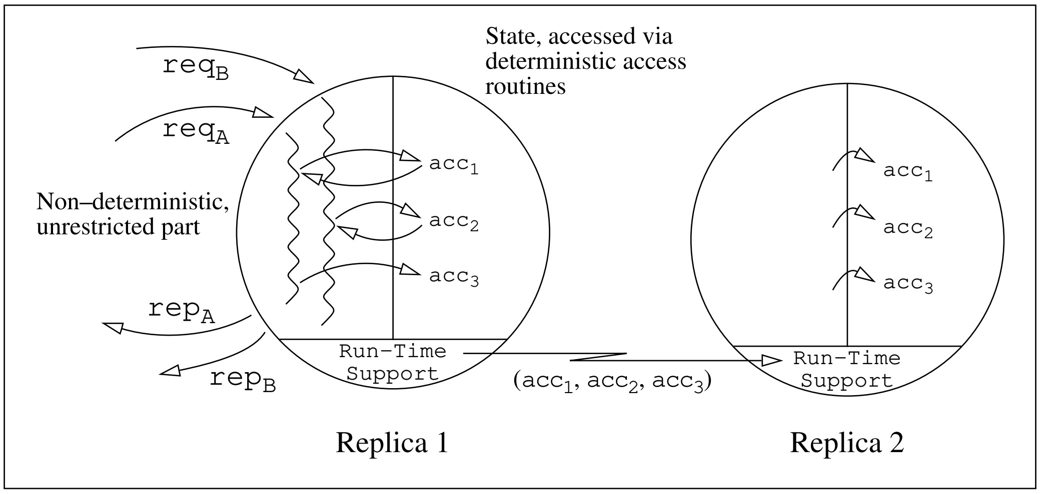


**Fig. 5.4:** Schema of Replicated Partitions

A partition is split in two parts: a general, unrestricted, non–deterministic part, and an encapsulated, deterministic part holding the coordinated state of the partition. Remote calls execute in the non–deterministic part and access the coordinated state only through a narrow interface. The replica manager enforces a common sequence of state accesses on all replicas. It does not, however, try to coordinate the task scheduling of concurrent remote accesses.

### 5.5.1 Organizing the Replicas

Replicas in the nondeterministic computation model can be organized using the coordinator–cohort protocol. Each (nested) RPC is handled as a (nested) transaction. Only the coordinator executes remote calls; the cohorts only replay the state accesses in the same sequence as they happened on the leader. Therefore, only the coordinator executes non–deterministically, while its cohorts only record the effects this non–deterministic execution has on the state.

There are close similarities regarding transactions between coordinator–cohort replication and databases: the cohorts assume the role of stable storage. For recovery, there are two cases to distinguish:

- The coordinator executing a request fails and one of the cohorts becomes the new coordinator and re–executes the request from scratch after having aborted all nested remote calls that already had been made by the request.
- The (replicated) client that had issued the request fails, and the new coordinator of the client instructs the coordinator that ran the request as a subtransaction to abort it.

The first case is similar to crash recovery in databases. During failure–free operation, the coordinator uses a deferred update scheme with respect to its cohorts: it sends them — upon the request's top–level commit or periodically — its intentions list, i.e. the list of state changes the request makes on the coordinator. When a top–level request is committed, the cohorts replay these changes to bring their states in synch with that of the coordinator.

The coordinator also informs its cohorts about any nested remote call it is about to make so that the cohorts are aware of them and can abort them should the coordinator fail before the request is committed. In this failure case, the cohorts discard the intentions list of the request, abort all nested actions, and then the new coordinator restarts the request from scratch.

The second case corresponds to abort recovery in database systems. The coordinator for a request must collect enough information to be able to abort the request at any time[1], even though it committed locally. Locally, the coordinator might use an update–in–place scheme, retaining before images of all objects modified by a request in an undo log until the top–level commit for that request arrives. If a request has to be aborted, the coordinator can use these before images to undo the request's (i.e., the subtransaction's) state changes. It also instructs its cohorts to abort the request, which makes them simply discard the request's intentions list since they didn't replay the intentions list yet. When aborting a transaction, the coordinator will also abort all its nested subtransactions.

So far, I have assumed that the coordinator is chosen statically: the same replica is the coordinator for all requests. This may be a good idea, because it automatically makes sure that once a subtransaction has committed locally, other subtransactions of the same top–

1. Until the top–level commit, at which point the undo log may be discarded.

level transaction can see the state changes. (Note that the cohorts update their state only once the top–level commit arrives.) It may, however, cause the coordinator to become a performance bottleneck as it has to do all the work while its cohorts are more or less idle. If the coordinator is to be chosen on a per–request basis to exploit parallelism and to get some load balancing, two modifications of the above scheme are necessary. First, cohorts have to replay a request's intention list as soon as the request commits locally on the coordinator, and they have to maintain undo information (before images) for this to support a possible later abort. In this way, the states of all replicas agree upon the modifications made by locally committed subtransactions, and other subtransactions of the same top–level transaction can be executed on any replica. Secondly, concurrency control, e.g. locking, must be synchronized between the replicas in a group: locks must be acquired in the same sequence on all replicas. The techniques described in [Bir85] can be used to optimize coordination of locks between replicas.

### 5.5.2 Identifying the State

In section 4.4 I have shown that a common sequence of state changes must be enforced among all the replicas of a partition in order to guarantee their consistency. In section 5.5.1 above, I have sketched a way to achieve this goal using coordinator–cohort replication by collecting an intentions list on the coordinator that is replayed on the cohorts. But what constitutes the state of a partition in Ada 95? Obviously, one cannot simply define the state as all global objects residing on the partition: global objects might be hidden in third–party libraries unbeknownst to the application or the replica manager in the run–time support. With such a definition of state, a transparent solution, i.e., collecting the intentions list without interacting with the application, is impossible.

It is therefore necessary to restrict the notion of "state" in some way. This is inevitably coupled with a loss of transparency because the application must somehow either implicitly know or explicitly specify what is to be considered "state".

I chose to breach the transparency rule using a new pragma called `Replicated` that would apply to packages [Wol97]. While clearly not transparent, this method integrates well with the handling of other aspects related to distribution in Ada 95: interface units of a partition already must be marked by one of the categorization pragmas. Using the new pragma, the application (or rather, its developer) can clearly identify which objects are crucial to the partition and must be synchronized among the replicas. If the partition is not replicated, a `Replicated` package will behave exactly like a normal package.

Basically, a `Replicated` package encapsulates global objects that are accessible only through access subprograms and entries of protected objects declared in the visible part of the package's specification. The compiler could easily recognize these subprograms and transparently add the necessary calls to the replica manager in run–time support to log them in the intentions list — this transformation is very similar in structure to the way RPC stubs for `Remote_Call_Interface` packages are generated. The `Replicated` pragma requires that the visible part of the package's specification contain only protected objects and subprograms. Furthermore, the body of a `Replicated` package must be *deterministic* (otherwise, replaying the coordinator's intentions list on the cohorts might give different results); i.e., the body must not contain tasks or task types, nor protected objects with entries other than those declared in the specification, nor any semantic dependencies on `Ada.Calendar` or `Ada.Real_Time`, nor any `delay` statements. It also shall not contain asynchronous `select` statements. Finally, a `Replicated` package shall depend semantically only upon `Pure` or other `Replicated` packages, because the parameters to the access routines must also be logged in the intentions list, requiring marshaling, which is defined only for types declared in packages categorized by one of the distribution pragmas. Dependencies upon `Remote_Type` packages are forbidden because their bodies are unconstrained [ISO95, E.2(9)] and thus may behave non–deterministically.

With all these constraints, `Replicated` packages cannot provide the mechanisms needed to handle replication correctly in the context of distributed objects (see section 3.2.4). Expanding their definition to encompass this concept seems not a good idea: `Replicated` packages are *internal* to a partition while `Remote_Types` packages by their very nature are part of the interface of a partition. As mentioned above, `Remote_Types` packages may be non–deterministic, making them useless for this approach of handling replication. A second new pragma `Replicated_Types` is needed, which has the same restrictions as `Remote_Types`, but additionally requires that the body be deterministic. Again, the invocations of primitive operations through remote access–to–class–wide types can be logged in the intentions list[1].

### 5.5.3 Drawbacks of this Approach

The above idea of splitting a partition in two parts and encapsulating the "state" in the deterministic part has some shortcomings in spite of its apparent simplicity. There are three main problems:

- The approach is not totally transparent, yet it doesn't integrate transactions as first–class objects in the language. This leads to some usage problems.
- An implementation of the scheme in a (nearly) transparent way would be very complex and difficult to realize.
- Since RPCs are supposed to become transactions transparently, deadlocks that may occur must be dealt with transparently.

1. I'll return to the problem of how to synchronize remote access values among the replicas in chapter 8. A similar technique could be used here.

Subsequently, I briefly discuss the first two points without going into too much detail because of the issue of deadlocks, which is discussed in the next subsection (5.5.4) and which made me abandon this approach altogether.

The main usage problem is that while specifying the state using pragmas integrates well into the standard model of distribution, using this state does not. An application designer must consciously decide which global objects shall be replicated, and concentrate all handling of these objects in packages that then can be marked with pragma `Replicated`. The package is a fairly coarse–grained tool for structuring the state; it might sometimes lead to unnecessarily complicated designs.

Further difficulties occur when distributed objects are used in a replicated partition. Although `Replicated_Types` packages provide a way to implement distributed objects that exist on all replicas, with primitive operations that are invoked on all replicas, their use is subject to some subtle conditions. First, derivations of a tagged type declared in a `Replicated_Types` package may only happen in other `Replicated_Types` packages — otherwise, the derived type might make use of non–deterministic features, which would destroy the state coherence of the replicas. Secondly, a partition that is to be replicated must not contain other distributed objects (i.e., with types declared in standard `Remote_Types` packages). Such objects would only exist on the coordinator because they are not replicated, which would cause major problems after a failure of the coordinator because the new coordinator might suddenly receive dispatching remote calls on distributed objects that do not exist. The application developer must exercise great care to make sure that a replicated partition uses only `Replicated_Types` packages for its distributed objects. Contrary to the first constraint on derivation, this condition could only be checked at configuration time.

Concerning an implementation of this scheme, there is an important number of intractable problems that have to be solved. Any call to a subprogram or an entry of a protected object declared in the visible part of a `Replicated` package or to a primitive operation of a type declared in the visible part of the specification of a `Replicated_Types` package constitutes a state access.

First and foremost, the transformation a compiler would have to do for `Replicated` or `Replicated_Types` packages is — except for toy examples — complex. The compiler would have to generate code to log each state access with the replica manager in the run–time support so that the latter can send this log to the cohorts to allow them to bring their state up to date. Objects encapsulated within `Replicated` and `Replicated_Types` packages must be recoverable, and it would be the compiler's job to automatically add the necessary data structures and calls to support packages. Basically, the types of the objects in the package must be rewritten to support recoverability. The compiler would also have to add the necessary support for concurrency control, e.g., by adding a transactional lock or timestamps to each `Replicated` package and protected object declared in the visible part of the specification as well as to each object of a tagged type declared in `Replicated_Types` packages.

Another major difficulty is hidden within the implementation of protected objects in the run–time support. Entries of a protected object not necessarily are executed by the task that originally made the call. If an entry call is queued, it may well happen that some other task executes the entry on the caller's behalf later on. Such an implementation may not work if 2PL is used as the concurrency control scheme and the transaction locks are not integrated with the implementation of protected objects. The implementation of protected objects in the tasking support must therefore be aware of transactional RPCs; the tasking system cannot be considered a "black box"!

There is yet another reason why this approach requires integration with the tasking system. The tasks that execute an RPC request must be identified: a transaction identifier (TID) [FHZ97] must be associated with each task. If a remotely callable subprogram itself creates subtasks, these too must get an identifier marking them as belonging to a particular transaction. One could use the standard package `Ada.Task_Attributes` to associate TIDs with tasks. However, there's no way to transparently get a particular task's parent, nor is there any way in standard Ada 95 to have a routine of the replication manager be invoked each time a task is created![1] The standard packages (e.g. `Ada.Task_Identification`) do not provide any way to find the `Task_ID` of the task that created a given task. Yet this information is known within the tasking system in the run–time support and the implementation of a replication manager has to access it, which necessitates an addition to the interface of the tasking system.

As far as a possible implementation is concerned, this approach requires elaborate cooperation of the run–time support with the code generated by the compiler. The transformations a compiler has to perform for `Replicated` packages are complicated and intricate, especially for adding recoverability to user–defined types. Adding concurrency control is only slightly simpler, and must be coordinated with the implementation of the tasking system. Yet these implementation difficulties can certainly be overcome, although the resulting system may not be elegant. However, by far the biggest problem of the idea to give RPCs the semantics of nested transactions is that deadlocks can occur.

### 5.5.4 Deadlocks

Assuming that the Ada 95 application has been written correctly and does not deadlock when partitions are not replicated, it must be guaranteed that replication does not cause new deadlocks to appear, or, at the utmost, that they can be dealt with in an appropriate manner, i.e., in a way that preserves the semantics of standard Ada 95.

1. Using a controlled task attribute added with `Ada.Task_Attributes` doesn't help: first, there's no guarantee whatsoever that the `Initialize` primitive operation of the attribute was called when the task is created because the attribute need not be created until the first access to it [ISO95, C.7.2(28)]; and second, there'd still be the problem of getting the parent task's `Task_ID` and TID to calculate the new task's TID.

The classical minimal deadlock situation in transactional systems arises when two transactions compete for access to the same two objects in the opposite order, as in the example in fig. 5.5 below.

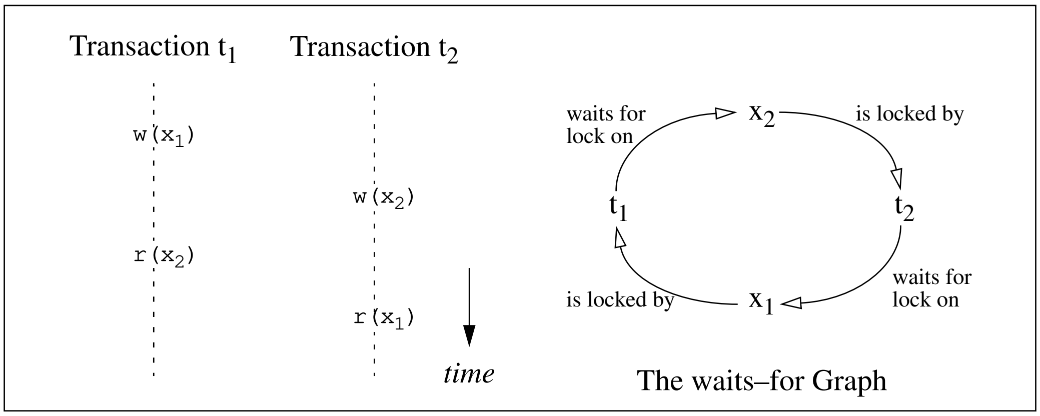


**Fig. 5.5:** Reader–Writer Deadlock

Here, two concurrent RPCs, i.e., nested transactions, access two protected objects $x_1$ and $x_2$. Transaction $t_1$ first writes object $x_1$ and then wants to read $x_2$, transaction $t_2$ first writes $x_2$ and then tries to read $x_1$. If the interleaving of operations is such that both transactions first perform the write operations, there's a deadlock between $t_1$ and $t_2$. Note that in Ada in general this does not result in a deadlock — it's the two–phase locking that causes it, because locks must be held until the end of the transaction to ensure strictness. There are several ways to resolve this simple deadlock:

- One of the two transactions could be aborted and restarted. This is the brute–force approach.
- A simple deadlock prevention scheme would be to require that all transactions lock all objects in the same order. Unfortunately, this would require that the compiler be able to analyze precisely which objects a certain RPC (transaction) will access, which in general is not possible as Ada 95 allows the dynamic creation of objects.
- A timestamp–based protocol would not run into a deadlock, but would abort and restart one of the transactions anyway.
- A multi–version protocol (of either the timestamp ordering or the locking variety) could be used. This in fact would avoid this particular deadlock without aborting either transaction, but (assuming multi–version timestamp ordering and $t_1.ts < t_2.ts$) transaction $t_1$ would not read the value $t_2$ wrote into $x_2$, but the value it contained before $t_2$'s write operation $w(x_2)$.

The protected objects of Ada 95 not only allow read and write accesses, but also conditional entries. This gives rise to a new class of deadlocks that are not detectable by looking for a cycle in the waits–for graph. An example is given in fig. 5.6, where transaction $t_1$ accesses a protected object $x$ through entry $e$ (and is granted a write lock on $x$ at the same time).

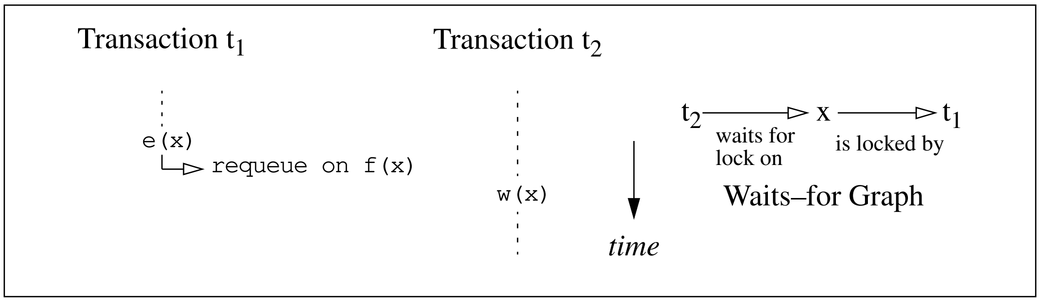


**Fig. 5.6:** Deadlock between a Writer and an Entry Call

The current state of *x* does not permit the operation on *x* to be performed, so *e* requeues on another entry *f* of the same object *x* whose barrier will become true as soon as the state of *x* changes such that the operation can be performed. Transaction $t_2$ then wants to do the write access to *x* that would do exactly this state change, but it will remain blocked because $t_1$ still holds the write lock on *x* it acquired upon entering *e*. (Note that the lock cannot be released upon requeueing, as this would violate the conditions of strict 2PL.) Again, normal operation of an Ada 95 programs does not deadlock in this situation because the mutex of a protected object is released when the requeue is done.

A multi–version protocol doesn't help at all in this case — $t_1$ *has to* wait for $t_2$'s write access, one cannot let it access an "old" value. The only solution to resolve this deadlock would be to abort $t_1$ and restart it later (after $t_2$ has done its write access); alas, the waits–for graph has no cycle, therefore a traditional deadlock detector wouldn't even find this kind of deadlock!

One could try to resolve this by augmenting the basic waits–for graph by adding edges that express waiting on an entry queue for the barrier to become true, yielding the graph shown in fig. 5.6a for the above example. This augmented graph *does* contain a cycle, which furthermore involves only transaction $t_1$, so $t_1$ would be (correctly) aborted. Once restarted, $t_1$ could execute normally once $t_2$ had committed, possibly even without requeueing on entry *f*.

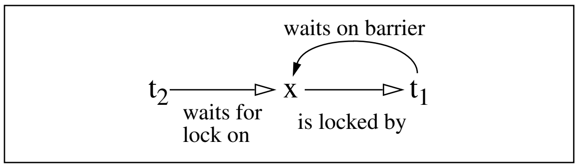


**Fig. 5.6a:** Augmented Waits–for Graph

Timestamp ordering could also be used to avoid the deadlock. Assuming $t_1.ts < t_2.ts$, progress will be made until $t_1$ tries to enter *f*: at that moment, $t_1$ is aborted because $t_1.ts < x.wt$ (see def. 5.8). When $t_1$ is restarted, it gets a new timestamp $t_1.ts > t_2.ts$ and therefore can execute normally, possibly even without requeueing on *f*. Note however that *strict* timestamp ordering *would* deadlock: $t_1$ would be waiting on *x*'s entry queue while $t_2$ would be delayed until $t_1$ had committed, which it could do only if $t_2$ were allowed to proceed.

The combination of the cases in figs. 5.5 and 5.6 results in a program structure that dooms the whole approach. An example is shown in fig. 5.7,which basically has the same structure as fig. 5.5 except for replacing the read accesses with entry calls whose barriers

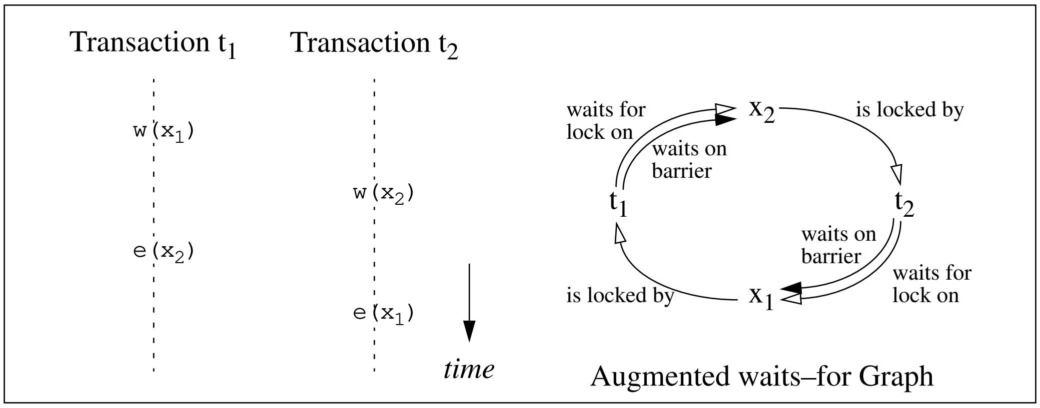


**Fig. 5.7:** Unresolvable Deadlock

can only become true if the write access of the other transaction is first executed. Assuming the interleaving of operations shown in fig. 5.7, both transactions first execute the write access. 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 locks the object. However, it will remain blocked 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!

A similar situation would arise if strict timestamp ordering was used instead of 2PL. Assuming that $t_1.ts < t_2.ts$, $t_1$ would be aborted when it tried to make its entry call $e(x_2)$ (because $t_1.ts < x_2.wt$, see def. 5.8 above). The other transaction $t_2$ could then proceed and make its entry call $e(x_1)$, which would block immediately on its barrier because $t_1$'s write access was undone. Transaction $t_1$ would be restarted with a new timestamp $t_1.ts > t_2.ts$ and execute its $w(x_1)$ operation, causing $x_1.wt$ to become $t_1.ts$. Now transaction $t_2$ could basically enter $e(x_1)$ as $t_1$ just changed the state such that the barrier became true, but it would be aborted immediately because now $t_2.ts < t_1.ts$... and so on, ad infinitum. Once again, the above situation works perfectly well in standard Ada 95.

## 5.6 Evaluation

In order to guarantee the consistency of the state of replicated Ada 95 partitions it is necessary to make all replicas execute all state accesses in the same order. Partitions in Ada 95 inherently are multithreaded, and hence the order in which requests start executing is of no importance. To avoid a costly domino effect, concurrent request executions must have the semantics of transactions.

The aforementioned implementation difficulties aside, a transparent implementation of such an approach is not feasible. Even using transactions, the effects of a failure are not local to the group of replicas as nested remote calls need to be undone. This forces a partition to maintain undo information to be able to roll back already executed requests. Even non–replicated partitions must execute remote calls as transactions if they are invoked from a replicated partition!

A further intrinsic problem with any transparent implementation of RPCs as transactions are long or unrelated subtransactions. The semantics an application implements in a transaction are sometimes such that the action need not be a properly nested subtransaction. Virtually all transaction systems offer a way for the application to define that certain nested transactions should actually be new top–level transactions. This mechanism can considerably improve the efficiency of nested transactions. In the approach presented in this chapter, it is not possible to provide such a feature without violating transparency even more.

The final and fatal problem of the approach has been discussed in section 5.5.4 above: transactional concurrency control may lead to deadlocks. The interactions of the semantics of the programming language Ada 95 and serializability are such that there may occur deadlocks that no concurrency control scheme can resolve. The fundamental issue is that the consistency criteria of the programming language and of transactions are incompatible. The protected objects of Ada 95 implement linearizability (cf. section 2.3.1) trivially by mutual exclusion while transactions require serializability as the consistency criterion. Serializability, being a blocking property, places much more rigorous restrictions on concurrency than linearizability does. Deadlocks may occur even if all operations are total — however, these can be dealt with by aborting and restarting one of the transactions involved. But the existence of partial operations (i.e., conditional entries of protected objects in Ada 95) may result in deadlock situations that are impossible to resolve and *that do not occur when the partition is not replicated!* If RPCs are transparently turned into transactions, innocuous code that works perfectly well when a partition is not replicated might suddenly block when it is.

The application must be aware of this possibility and be written in such a way that these unbreakable deadlocks cannot occur. This is an implicit constraint that may be rather difficult to adhere to without the help of sophisticated development tools. I see two possible remedies for this severe problem:

- Transactions can be offered on the language level, in a non–transparent way, or
- one allows only protected objects without entries in the state of a replicated partition.

The first is certainly a possibility worth exploring, but runs counter to my goals of offering fault tolerance by replication in a *transparent* manner. Still, if transactions were introduced as a construct visible to the application, deadlock situations of the type shown in fig. 5.7

above would be acceptable (albeit unfortunate): the developer would be using transactions consciously in this case, and taking care of the potentially problematic interactions between entry calls on protected objects and transactional serializability was thus the application's responsibility.

The second is a rather brutal and unsatisfactory fix: if no entries of protected objects in the `Replicated` packages are allowed, these become awfully similar to standard `Shared_Passive` packages, and the implementation would be scarcely more than a complicated and contorted way to implement a distributed shared memory.

For all these reasons I finally decided to drop this approach and to pursue instead the approach presented in chapter 6.

# Chapter 6: Piecewise Deterministic Replicas

## 6.1 Non–Determinism

In the preceding chapter, I have shown that replication under the assumption of the non–deterministic computation model of the Ada 95 language standard [ISO95] must be based upon transactional semantics for remote operations, and that this cannot be achieved in a transparent manner due to incompatibilities of the consistency models of Ada 95 (linearizability) and of transactions (serializability).

It seems therefore that the non–deterministic computation model leads to a dead end as far as transparent replication is concerned. Can we assume another computation model that would offer a way to side–step the problems encountered in chapter 5?

What are the fundamental causes of non–determinism in Ada 95? The list in section 4.4 deals only with language constructs and their definitions. The non–determinism perceived at the language level is caused ultimately by *deliberate underspecification*. The language standard *abstracts* from particular implementations. Abstractions are a way to focus on important issues, to mask details of a system that are deemed not important for the purposes of the discussion at hand — in this case, for the specification of the programming language.

On the language level, there's no way to avoid this non–determinism. One can, however, descend at a lower level of abstraction and contemplate the problem on the level of the run–time support of Ada. In chapter 5 I discussed a solution at the abstraction level of the language, in particular, I considered the run–time support as a black box, except for the PCS. In this chapter, I will assume a white–box view of the run–time support and especially of the tasking support. As I will show, this indeed yields a model that is far better suited for transparent replication of Ada 95 partitions.

The other fundamental reason for non–determinism in Ada 95 (besides deliberate underspecification) is the implicit dependence on time. As I have shown in section 4.4.2, it is not possible to exorcise this source of non–determinism. However, the semantics of Ada 95 are such that the behavior of correct applications is independent of timing. Here, the abstractions made in the language standard provide a way to circumvent the timing issue altogether.

## 6.2 The Computation Model

I assume the full computation model defined in the Ada 95 language standard. I only assume that the execution of a partition consists of a sequence of *piecewise deterministic state intervals* [SY85, Eln93]. Whenever a *non–deterministic event* occurs in the partition, a new deterministic state interval is started. The goal of replication management is to guarantee that all replicas go through the same sequence of state intervals, which clearly implies that the same non–deterministic events occur in the same order *and at the same logical moment* on all replicas.

A non–deterministic event in an Ada 95 partition can be either an *external event* such as the delivery of an RPC request message sent by some other partition or an *internal event* such as a task scheduling decision. Each task within a partition itself follows a piecewise deterministic execution path. The nondeterministic choices made whenever one task interacts with another one separate the state intervals of the tasks. The execution of the whole partition can be seen as a set of parallel sequences of state intervals, one sequence for each task.

### 6.2.1 Validity of the Model

The abstractions in the language standard guarantee that a correct Ada 95 application is independent of the precise moments tasks access the state; the result does not depend upon the precise interleaving of executions.

Instead, global state that is to be accessed concurrently by more than one task *must* be protected by appropriate synchronization in Ada 95 (e.g., through protected objects): accesses to one object *always* happen in some sequential order. A program execution in Ada 95 is deemed *erroneous* if two tasks ever try to access the same unprotected object simultaneously. More precisely, the language standard requires that accesses be sequential, i.e., actions of different tasks on the same objects are only allowed if the action of one task *signals* the action of the other task. [ISO95, 9.10(3–10)] lists the precise conditions under which signaling in this sense occurs.

In summary, global state in Ada 95 must be either encapsulated in protected objects, or be protected against concurrent accesses by appropriate application–level synchronization using rendezvous; otherwise the application is erroneous! Without loss of generality, one may assume that any global data is encapsulated in protected objects[1]. Because protected objects implement linearizability (cf. def. 2.2; mutual exclusion is the basic implementation of a sequential specification), and because linearizability is a local property, the execution of the whole application is linearizable. Any interleaving of piecewise deterministic state intervals that will occur in a correct Ada 95 application will yield a global linearizable history.

The net result of the signaling semantics of concurrent execution of Ada 95 applications is that only the observed execution history is significant, irrespective of timing. Therefore, synchronization of replicas using the piecewise deterministic model can be achieved by recording this history on one replica and forcing all other replicas to follow the same history. That way, timing differences between the replicas that may occur due to the absence of a common time base have no effect on the correctness of the execution.

## 6.3 Semi–Active Replication

The semi–active replication model [BHV+90] has been introduced in the Delta–4 project [Pow91] to overcome deficiencies in both the passive and the active replication models: in the former, there's an important recovery latency after a failure; the latter requires deterministic replicas. Semi–active replication supports non–deterministic replicas while offering an availability nearly as high as active replication.

In the semi–active (or leader–follower) model of replication, all replicas execute incoming 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 take the same decisions.

The Delta–4 Extra Performance Architecture (XPA) [VBB+91] uses the notion of *pre–emption points* to coordinate replicas in the leader–follower model. These are predefined points in a replica at which pre–emption may occur. The leader sends a synchronization message to its followers each time it

- delivers a request, and
- reaches a pre–emption point.

Followers are always executing one step behind the leader; i.e., one synchronization message behind it. In Delta–4's XPA, requests are multicast to the group using reliable, not totally ordered multicast. Synchronization messages from the leader to the followers also

1. 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.

use reliable multicast only, ordering is implemented in a higher layer. (Synchronization messages contain enough information so that the followers can detect and wait for missing messages.)

Semi–active replication is based on the notion of view–synchronous communication (see section 2.3.3). Without view synchrony, different followers might deliver a synchronization message in different views. An example showing how this could lead to inconsistencies between replicas is given in [Maz96, pp. 86f].

Contrary to the coordinator–cohort model, where a new coordinator can be chosen for each request, the leader is chosen statically: at any time, there is only one leader in a group of replicas; it is the leader for *all* requests. Two requests may interact through internal events, and the precise nature and sequence of this interaction is determined on the leader.

## 6.4 Replica Management

Global data that is accessed concurrently must be protected by protected objects in Ada 95 (or declared atomic using the pragma `Atomic`, but I will not consider this rather special case). For state accessed (directly or indirectly) by remotely callable subprograms this means that *all* the state must be contained in or at least be protected by protected objects because even two calls to the same remotely callable subprogram may execute concurrently.

### 6.4.1 Events

Deterministic state intervals are demarcated by the occurrence of non–deterministic events. I distinguish external and internal events.

*External events* occur at any interaction of the partition with the rest of the system and account for non–determinism in the communication support. An external event can be:

- The 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.
- Reception of an RPC answer message from some other partition.

*Internal events* are all non–deterministic events that can occur *within* a partition. They cover all task dispatching points [ISO95, D.2.1(4)], signaling actions, and events related to abort deferral. This includes:

- 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 of 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.
- Entering and leaving the `Initialize` or `Finalize` primitive operations of `Controlled` and `Limited_Controlled` types.
- 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. I will discuss this kind of event in detail in section 6.5.

Logging these events on the leader and replaying them on the followers in the same order is sufficient to guarantee that all replicas evolve in the same manner. It is not necessary to synchronize the replicas down to the level of machine instructions (impossible in a heterogeneous system) or simple Ada statements — which might be arbitrarily complicated if not impossible: consider for instance a partition running on a multiprocessor, where different tasks might execute on different processors.

### 6.4.2 Coordinating the Replicas: Observable Events

Semi–active replication has the advantage that no global order on incoming requests must be imposed on the communication protocol level at the replicas. Since replicas coordinate the order of *events*, it would make no sense to impose an ordering on *requests*. Consistent ordering is only imposed on the delivery of the synchronization messages the leader sends to its followers, whereas clients of a replicated partition may use a relatively simple reliable multicast primitive for sending requests instead of a much more complicated and expensive totally ordered multicast. The replicas themselves, however, must agree on the order of events, which could be implemented similarly to the approach used in Delta–4 [VBB+91, pp. 244f] or by using a FIFO multicast for intra–group synchronization[1].

Multicasting a synchronization message within the group each time an event occurs most probably 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

1. FIFO order is sufficient because only the leader ever sends synchronization messages.

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. I 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. 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.

### 6.4.3 Correctness of the Approach

In section 6.2.1 I have already shown that the correctness of Ada 95 applications does not depend on implicit timing dependencies such as the precise interleaving of executions of concurrent tasks. The signaling model of concurrent accesses to objects in the global state defined in [ISO95, 9.10(3–10)] has the effect that all such accesses to the same object effectively happen in some sequential order, separated by signaling actions. The set of internal events defined above encompassing all these situations, the execution histories of all replicas will be identical to the history observed on the leader.

If a partition violates the signaling model by accessing an *un*protected object in different tasks, event logging cannot guarantee that all replicas perform the accesses in the same order. Since a replication scheme must only ensure consistency for *correct* partitions, it is quite acceptable to have replicas of a partition that does contain unprotected accesses to shared objects diverge. Replicating an erroneous partition will not make it correct!

This leaves only modifications of state local to a task to be considered. Given two tasks that each modify a local object, it is immaterial which access happens first on a global "wall clock" time scale. The two accesses cannot possibly affect one another, for they happen in different state intervals of different tasks during which there is — by definition — no communication between the tasks. If any task synchronization finally occurs, this constitutes a non–deterministic event, terminating the state intervals. Logging and replaying of the events in the same order will guarantee identical evolution of state intervals on all replicas.

Asynchronous abortions of either tasks or sequences of statements (cf. section 3.1.4) might basically cause the replicas' states to diverge if the abortion didn't happen at precisely the same logical moment during execution.

For instance, in the asynchronous `select` statement shown in fig. 6.1, the abortable part 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 (after `Do_It_1`, or after `Do_It_2`, or even *during* `Do_It_1` or `Do_It_2`, or during the evaluation of their parameters), 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.

```
select
  Trigger.Wait_To_Occur (...);
then abort
  Do_It_1 (...);
  Do_It_2 (...);
end select;
```

**Fig. 6.1:** Asynchronous `select` Statement

Internal events correspond to the concept of pre–emption points in Delta–4. True task pre–emption (time–slicing; or even true parallelism in the case of tasks executing on different processors of a multi–processor system) does not enter the picture at all, because consistency is defined purely by the visible effects an execution has on the state. Given the language rules on signaling, abortion, and abort deferral, coordinating state accesses by replaying the observed event history is sufficient to guarantee consistent state evolution.

### 6.4.4 Failures

Failures in the leader–follower model of replication have to be handled differently depending on the role the failed replica had assumed. When a follower fails, nothing has to be done — except possibly reconfiguring the distributed application by starting a new replica to restore the original replication degree (see section 6.4.5 below). 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.

Upon a failure of the leader, one of the followers must become the new leader. The new leader 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. 6.2.

The leader $R_1$ of a replicated CS–component started executing a request $\mathtt{req}_A$, made a nested remote call $\mathtt{req}_B$ to some other partition P, waited for the result, and continued processing $\mathtt{req}_A$ before it finally failed. Just before it performed $\mathtt{req}_B$, it sent its event log for the extended state interval $S_1$ to its followers $R_2$ and $R_3$, which replay the log to execute $S_1$. When $R_1$ fails, a view change occurs and follower $R_2$ 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 $\mathtt{req}_B$, which is guaranteed to be identical), but since $S_2$ could not possibly have affected any other part of the system except $R_1$ itself, the application's overall state is still consistent.

Note that in the example in fig. 6.2, I assume that communication with a group of replicas is always by reliable multicast: even the reply $\mathtt{rep}_B$ is multicast from P to all replicas. $R_2$ therefore doesn't need to re–execute $\mathtt{req}_B$ at the very beginning of $S_3$ if the answer already arrived. If the view change due to the failure of $R_1$ had occurred *before* $\mathtt{rep}_B$ had arrived, $R_2$ could *not* just wait for the reply: the former leader $R_1$ might have failed at point $\pi$, i.e., after having synchronized the extended state interval $S_1$, but before having started $S_2$

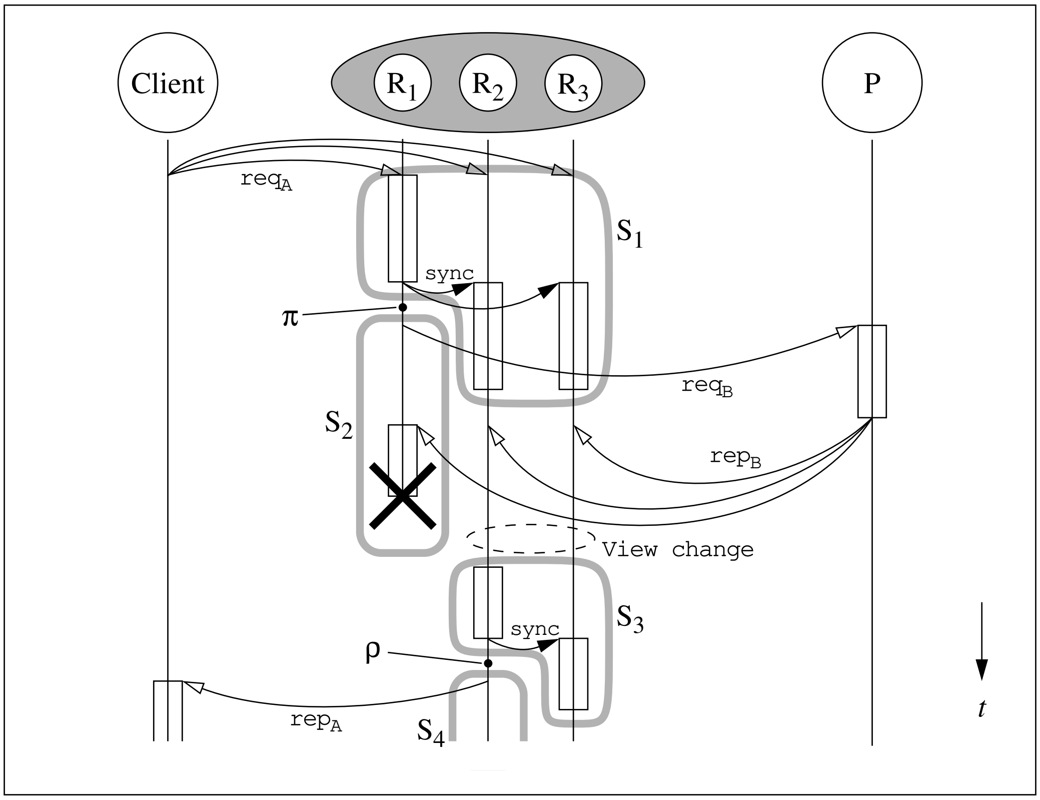


**Fig. 6.2:** Failures in Leader–Follower Replication

by sending the RPC request `req`$_B$. Instead, the new leader $R_2$ 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 `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 $R_2$ fails after point ρ, $R_3$ will become the new leader. It cannot know whether or not $R_2$ already sent back `rep`$_A$ to the client, so $R_3$ 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. 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.

Duplicate message detection cannot be avoided unless both the leader, all its followers, *and the destination partition* can agree whether or not a particular message has been sent and delivered. This would be the case if they all were part of the same group, so that a request or answer message could be sent atomically by a reliable multicast to the destination partition and to all replicas of the sender at the same time. For this to be possible, both partitions would have to be in group: a group would have to be created for each pair of communicating partitions. Given *n* partitions, this would require the creation of $O(n^2)$ groups in the worst case where any partition communicates with all the $n-1$ other partitions. Furthermore, such a grouping would increase network traffic considerably because the destination partition would not only get the RPC request or answer message addressed to it, but also all intra–group synchronization messages, even though those are significant only for the followers — the destination partition can only discard them.

What would be needed is a group communication primitive supporting atomic sending of messages to multiple (two) groups [SR96]. This paper proposes a primitive for totally ordered multicast of a message *m* to several groups. The message *m* in this case would contain the request for the destination partition and the extended state interval for replica synchronization. A minor deficiency would still be that the destination partition would also get this synchronization information, which is meaningless for it. Maybe a primitive for sending two messages to two groups atomically would be more appropriate[1].

### 6.4.5 Recovery

The method of organizing replicas described so far ensures the consistency of followers when the leader fails. Unfortunately, the replication degree decreases with each failure. It is therefore highly desirable to be able to add new followers to the group by restarting replicas of the partition to replace failed ones.

Once a new replica has been elaborated and started, its first interaction with the rest of the system will be to join the group of replicas. This leads to a view change in the group, installing a new view including the new follower. The group composition is therefore *dynamic*. As part of this view change protocol, the new replica must obtain the current state of the other group members. In this subsection, I'll explore the question of how to organize this *state transfer*.

In a *homogeneous* system, the state transfer amounts to taking a system–level checkpoint on one of the replicas that also were in the previous view and installing this checkpoint on the new replica, which then can resume execution in a state that is consistent with the rest of the group. The system checkpoint is a snapshot of the internal state of the replicas: it includes the data space, but also all relevant system information such as program counter, stack, task states, and so on. Note however that the state transfer cannot simply be done by installing a memory image of one of the current followers in the new replica. This

1. Note that a reliable multicast implementation already needs to do duplicate message detection. 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.

would not be entirely correct: any reference to physical devices such as communication channels, terminals, or files must be recreated in a meaningful and consistent fashion on the new replica; furthermore, if files are not shared, any file that has been written to must be included in the state.

I will not discuss how to take such a system checkpoint: this is beyond the scope of this thesis. Checkpointing multithreaded processes is a research topic in its own right. The well–known `libckpt` checkpointing library for Unix [PBKL94] is limited to sequential programs. The Condor process manager [LTBL97] for Unix systems is limited to single processes that do not communicate with other processes (though it is being enhanced recently, see [PL96]). A more recent development called STAR [SF98] assumes piecewise deterministic behavior of processes, but assumes that the only non–deterministic events are message deliveries and hence is also limited to single–threaded processes.

In *heterogeneous* environments, such system–level checkpoints cannot serve to implement the state transfer. The system–specific data they contain is meaningless on a different physical node unless nodes were homogeneous. Yet the new replica must somehow be brought up to date with respect to the other replicas in the group. There are two problems in organizing the state transfer:

- How to identify the state, and how to collect and install it?
- How to guarantee that the new replica starts executing at the right point in the flow of control?

In a heterogeneous system, dynamic groups are feasible only for a restricted subset of all possible partitions, as I will show below: only partitions where all activity is directly triggered by incoming RPC requests and where all tasks that are created by the application (not by the run–time support) eventually complete, and where elaboration of the partition's library units does not create tasks, can be replicated dynamically. For all other partitions, only static replication is feasible in a heterogeneous environment.

Given that a true checkpoint cannot be taken in a heterogeneous environment, there is no way to transfer the state transparently. The only solution is to have the application itself collect and install the state. A partition that is to be replicated must offer two subprograms called `Get_State` and `Put_State`. The run–time support invokes these subprograms at appropriate times during the view change to implement the state transfer. `Get_State` is invoked on one of the old group members to collect the group's current state, which is then sent to the new replica. On the newly joining group member, the run–time support then calls `Put_State` with the just received state. `Put_State` should then install this state in the new replica. Subsequently, I will use the term "checkpoint" to designate such an *application–level checkpoint* taken by `Get_State`.

The `Get_State` and `Put_State` subprograms may be far from trivial. `Get_State` must traverse the whole state of the application and marshal it, even if some objects do not have remote types (pointers in particular!). In this case, the application must implement its own, representation–independent encoding. It also must deal properly with third–party libraries whose source code may not be available; such libraries must be encapsulated in wrapper packages that do store enough information that will be collected by `Get_State` such that `Put_State` can recreate an equivalent state. If the partition modified any files, they must be considered part of the "state", too, and `Get_State` must include information in the checkpoint allowing `Put_State` to bring the file on the new replica up to date.

The second question is much harder. There is no way the new replica can resume execution at an *arbitrary* point in the control flow for this implied that one could actually take a system checkpoint including tasking state, stack, program counter, and so on. The new replica can only start executing requests from the beginning. Unless corrective measures are taken, this may lead to inconsistencies as is illustrated in fig. 6.3.

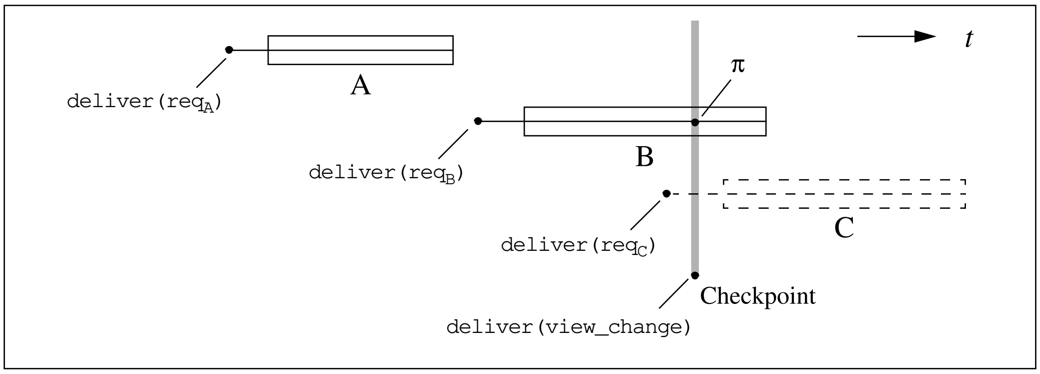


**Fig. 6.3:** Collecting the State

In this figure, the leader replica of a replicated partition is shown. It is executing three requests A, B, and C concurrently, each in its own task, when a view change occurs because a new replica joins the group, and the state must be collected using `Get_State` and transferred to the new replica. The three requests shown are exactly the three important cases to consider. Request A already has completed by the time the view change occurs: its reply has already been sent to the client. All its effects are reflected in the state, and the new replica doesn't have to do anything for this request. Request C has not yet been started, although the request message has been delivered before the view change. Again, the new replica has nothing to do for this request, as it didn't have any effect on the collected state yet. However, the request message for C must be forwarded to it because this request will affect the state, but the new replica didn't receive the request message as it wasn't yet a member of the group. This would mean that later on, the new replica would receive synchronization messages from the leader containing events involving a request it didn't know about!

Both these requests are easy to handle correctly. Request B, on the other hand, is more problematic as it is in progress when the view change occurs. A first problem is that since the request is active, it may hold resources, i.e., it may currently be active within a protected action. Since `Get_State` is an application–level subprogram, it must acquire read access to these resources, too. This implies that the run–time support must delay the invocation of

`Get_State` until all currently active requests have reached points where they do not hold exclusive access to some global data item.

Furthermore, the new replica, being unable to resume execution of this request precisely at point $\pi$, can only restart B from scratch. However, the state it receives already contains B's effects up to $\pi$, and re–executing it will therefore make these state changes twice on the new replica, which may cause its state to diverge from that of the other replicas. Read accesses prior to $\pi$ that are re–executed on the new replica may return the wrong values, since the state at $\pi$ may be different than the state on the leader at the time the read access was executed originally. How to solve this problem?

A first approach is similar to the one taken in chapter 5 for the state in the transactional model of RPCs. The state must be identified somehow (e.g., using the pragma `Replicated`), and the replica manager in the run–time support not only logs events, but for any state access, it also logs the returned values (if any). This log is sent together with the state to the new replica, which can then re–execute request B from scratch, returning the logged results of B's state accesses prior to $\pi$ instead of re–executing them. The major drawback with this scheme is that "state" would have to be restricted nearly as much as in chapter 5, in particular, types used in the interface to the state would all have to be remote types because to include them in the log, they must be marshaled. This would require elaborate support from the compiler to insert instructions to marshal the results and log the accesses.

Another approach is to assume that all RPCs eventually complete and to wait for *quiescence* before collecting the state, i.e., to delay calling `Get_State` until all currently active requests (like request B in fig. 6.3) have completed. Requests that arrived before the view change but that didn't start yet are queued and will be started only once the checkpoint has been taken. This situation is shown in fig. 6.4.

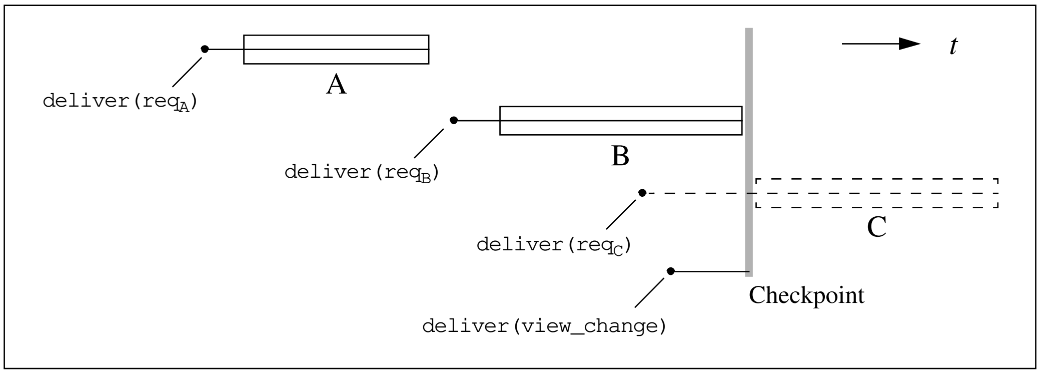


**Fig. 6.4:** Waiting for Quiescence

Collecting the state would be delayed until B has completed, and request C would be queued — its execution would be deferred until after the state had been collected. Once the state has been collected, it is sent together with all queued requests to the new replica. This avoids the problem at the cost of introducing an arbitrarily high latency for the join of a new replica. But again the synchronization mechanisms of Ada 95 complicate matters: request B might actually be waiting on an entry barrier of a protected object that will be set to true only by request C! In this case, the replica manager would wait in vain for a quiescent state as B would never complete.

Because queueing on an entry call is an internal event, the replica manager is aware of such blocked RPCs, though. It can basically detect this situation and decide to allow some of the deferred requests to execute in the hope that the blocked requests will be unblocked, delaying the checkpoint further until finally a quiescent state was reached. It might even make some “intelligent” guesses as to which of the deferred requests might be likely to unblock the blocked requests: by observing past execution history, it could note that certain remotely callable subprograms are likely to access certain protected objects, and could choose a deferred request for execution that usually accesses the protected objects other calls are blocked upon. This heuristic would increase the probability that blocked requests would be able to continue and complete soon. However, even such a heuristic could not *guarantee* that a quiescent state would eventually be reached.

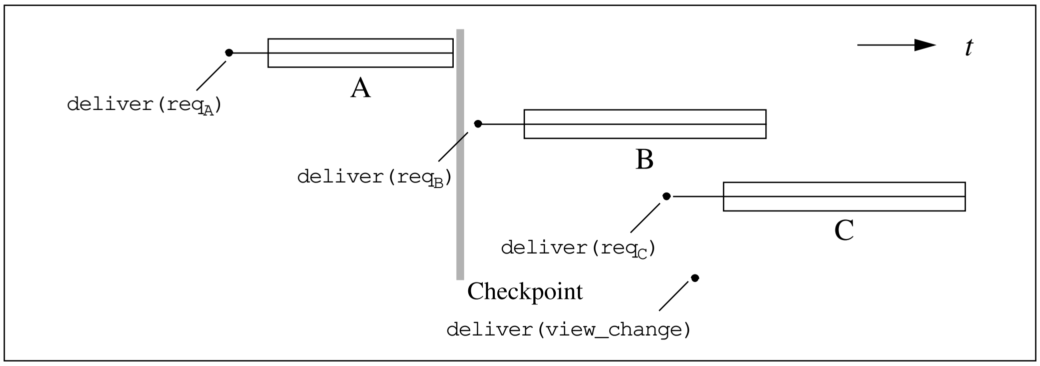


**Fig. 6.5:** Opportunistic Checkpointing

Yet another approach is to simply ignore the blocking problem as shown in fig. 6.5. A checkpoint would be taken whenever a quiescent state is detected. When a view change occurs, the run–time system might — in an attempt to minimize the data to be sent for the state transfer — try to take a new checkpoint, but if the situation was not quiescent, the new replica would simply get the last checkpoint together with the log of all extended state intervals since then, plus all messages delivered since then ($\texttt{req}_B$ and $\texttt{req}_C$ in the example). This simple scheme actually does not introduce any waiting for quiescence during a view change. The start-up latency of the new replica may be quite high all the same, since the log of all extended state intervals since the last checkpoint taken at a quiescent state may be arbitrarily long.

A severe problem with any checkpointing scheme based on the notion of quiescence are autonomous tasks within the replica, i.e. tasks that do not terminate. An example might be a global task that constantly updates some global object with a sensor reading. If such tasks are present, there would never be a quiescent state after the initial state and thus no checkpoint could ever be taken, causing the log of extended state intervals to become very long indeed over time.

Active tasks at the time a checkpoint is taken complicate matters anyway. Not only may they hold resources, but they also must be recreated on the new replica. In general, the run–time support has not enough information to accomplish this feat. For instance, it doesn't even know where the code for a task's body is! The compiled code passes a reference to the body to the run–time support routine for creating a task all right, but this reference would be meaningless on the new replica. Also, even if a task could be recreated correctly, the problem of resuming execution at the right point in its flow of control remains. Logging all its actions and replaying this log on the new replica is impractical: first, this log may be arbitrarily long, and second, it may contain events based upon interactions with tasks that do not exist anymore.

Global tasks created during the elaboration of a partition may cause problems of a somewhat different nature for recovery. In the absence of tasks, elaboration is a sequential process performed by the environment task. If elaboration of the partition creates global tasks, these may interact with each other. When a new replica is elaborated, such tasks should therefore be started. But they might work with the wrong data: `Set_State` can be called only once the partition has been elaborated, and state transfer requires that tasks be started (or replayed) only once the new state has been installed.

For all these reasons, transparent state transfer in the presence of arbitrary tasks is not possible in a heterogeneous environment. For an implementation, I will therefore make the following assumptions:

- State transfer is implemented by taking a checkpoint calling an application–level subprogram `Get_State`. This checkpoint is installed on the new replica by another application–level subprogram `Put_State`.
- If the state contains any objects with types that are not remote types, these subprograms must encode them appropriately in order to be able to recreate these objects again in a heterogeneous system.
- I also assume that replicated partitions are S– or CS–components. All activity is triggered by the delivery of remote requests.
- Elaboration of the partition does not create tasks, and any tasks created directly or indirectly through a remotely callable subprogram eventually complete.
- The replica manager supports dynamic groups, i.e. recovery of replicas, only under the quiescence assumption: from time to time, there is a moment at which no remotely callable subprogram is active and at which a new checkpoint can be taken. A new replica gets the last checkpoint plus the event logs for all extended state intervals since then.

Recovery is the one aspect of replication that is not transparent in my approach. Transparency is only maintained for static groups, i.e., groups where replicas can only fail. With this restriction, semi–active replication of partitions for fault tolerance offers *k–resilient objects* [LS90], i.e., objects that can tolerate up to *k* failures.

The only related work for general, portable checkpointing for heterogeneous environments I am aware of is `c2ftc` [RS96, RS97] and its successor `porch` [Str98]. `c2ftc` uses source–to–source compilation of C programs, heavily instrumenting the original code to maintain a portable shadow version of the memory (both stack and heap). Checkpoints must be manually indicated in the original source code by calling a system function `checkpoint`. `c2ftc` automates taking and installing checkpoints across binary incompatible platforms, i.e. it does the work supposed to be done by the `Get_State` and `Set_State` subprograms in my more traditional approach. However, C being a purely sequential programming language, `c2ftc` does not deal with problems arising due to multithreading. Its successor `porch` is intended to function within the context of a programming language called Cilk, a multithreading extension to C developed at the MIT. Cilk's concept of multithreading is fairly limited, though, in particular, it doesn't seem to have a concept corresponding to entry barriers in Ada 95.

## 6.5 Interacting with the Real World

Interaction of replicated partitions with the real world is always problematic, as the real world normally does not have any of the "nice", well–defined properties of Ada 95 partitions. In this section, I discuss some of the problems.

### 6.5.1 Files

If replicas access files, the handling of file operations depends upon the nature of the file system. If each physical node executing a replica has its own, non–shared file system, I assume that they are identical in the beginning. In this case, nothing special has to be done for file accesses: as all replicas will follow the same sequence of state intervals as on the leader, they will all open the same files and perform the same operations on them in the same order. Therefore, corresponding files on the different file systems belonging to the different physical nodes on which replicas reside will evolve identically. Failures do not pose problems either: if a replica fails, the associated local file system becomes inaccessible for the rest of the Ada 95 application and the question of whether all files are in a consistent state becomes moot. Files can be seen in this case as an extension of a replica's state to secondary storage.

If all the physical nodes executing the replicas of a partition have access to one single shared file system[1], there are two cases. If the file server is part of the Ada 95 application,

1. I will not consider the case where some replicas of a partition share a file system while other replicas of the same partition have their own file systems.

i.e., if it is a partition of the application, all accesses to it automatically are remote calls. File system accesses are normal observable events, and their handling doesn't differ in any way from that of normal observable events. The implementation of the standard packages for handling files (`Ada.Text_IO`, `Ada.Wide_Text_IO`, `Ada.Direct_IO`, `Ada.Sequential_IO`, and `Ada.Streams.Stream_IO`) must cooperate with the replica manager in the PCS to log and replay these events. Like any partition, the file server must be able to deal correctly with duplicate messages.

If the shared file system is provided by an operating system, the situation changes somewhat. Access to such a file server is through normal *operating system calls* made within the standard Ada 95 packages for handling files such as e.g. `Ada.Text_IO` or `Ada.Streams.Stream_IO`, not through Ada 95 remote calls. This means that one cannot assume that the file server be smart enough to discard duplicate accesses — it has no way to recognize duplicate messages as such. It is therefore impossible in this case to ensure consistency of the file system in all cases; the replicas alone cannot always decide whether or not a particular access has been made. Cooperation of the file server is needed to enable the leader to perform the access to the file system *atomically* with the synchronization of the extended state interval. (See also footnotes 1 and 2 on page 94.)

### 6.5.2 Terminal I/O

Terminal I/O is very similar to file I/O. The user can be seen as a separate partition. Writing on the terminal corresponds to sending it a message, input from the terminal is equivalent to receiving an answer message. Note that this means that either the terminal is smart enough to handle duplicate messages as discussed in section 6.4.4 or that *stuttering* may occur: if the leader fails, a new leader may repeat some terminal outputs, and it may request certain inputs again.

### 6.5.3 Sensors and Actuators

Sensor readings present some interesting problems because their values usually are very sensitive to timing — which is the main reason for my not considering real–time applications. Because reading the same sensor at different times may yield different results (as might simultaneously reading different sensors that are supposed to monitor the same condition), followers cannot re–read the sensor. Sensor accesses must instead be considered system events: followers replay all sensor accesses by returning the results obtained on the leader[1].

This is exactly what is done for `Ada.Calendar.Clock` (which is also a sensor reading in a way). Note that if time as returned by `Ada.Calendar.Clock` should be monotonic, the

1. By returning an "old" value as the sensor reading, the follower is of course no longer synchronized with the real world. This is one of the reasons why I excluded real–time applications from my considerations.

time base [ISO95, 9.6(6, 23), D.8(34–36)] must be coordinated between the replicas. Clock drift or allocating replicas on physical nodes that lie in different time zones might otherwise cause time to "jump backwards" between two extended state intervals when the leader fails.

Actuators such as robot arms commanded by a replicated partition can be seen as another kind of output devices, similar to terminals or file systems. They too must be smart enough to handle duplicate messages, otherwise the device might be instructed several times to perform some action it had already done.

### 6.5.4 Recovering from a Failure

Failures of real–world entities are not of interest for discussing replication: even a non–replicated partition has to handle these cases gracefully on the application level. I therefore assume that if the file system, or the sensor, or the actuator fails, the application developer has foreseen this case and programmed appropriate recovery mechanisms into the application.

Unfortunately, failure of the leader is not transparent as far as "dumb" output devices are concerned. If a file server or an actuator cannot properly deal with duplicate messages, then the new leader cannot simply pursue its computation. It depends upon the semantics the application defines for the observable event what corrective action (if any) is necessary. In general, actions for which the application requires at–most–once semantics need corrective actions, while at–least–once actions may be simply repeated. Any corrective action must employ forward recovery — the real world cannot roll back. For example, if the output device is a cash dispenser and the observable event is a message instructing it to dispense $100, at–most–once semantics is probably desired: the new leader must only re–execute this event if the ATM didn't already dispense the money. Presumably, the application could query the cash dispenser's state to find out whether or not this has happened[1]. If the event was a command to close the lid of the cash dispensing slot, at–least–once semantics might be appropriate and the new leader might simply re–execute the event, thus closing the lid really well.

A generic, transparent solution for this problem seems not feasible, as there are just too many application and environmental dependencies. The implementation discussed in chapter 8 does not try to handle this case, but offers an interface for applications to define

---

1. Note the *race condition* here: if the former leader sent the request message for dispensing the money before failing, but the message didn't arrive yet at the cash dispenser by the time the new leader makes its query, it is still possible that the cash dispenser will dispense the money twice. This can be avoided only if the cash dispenser still is smart enough to ignore old messages, i.e., if both the ATM and the application use some kind of *application–level*[2] message sequence numbering or FIFO protocol. [SC91] doesn't even mention this problem.
2. For this reason it is not possible to use this approach to handle an OS–level shared file system: the messages sent between the local OS kernel and the remote file server are beyond the control of the Ada 95 application, and therefore this race condition makes it impractical to try to have the new follower determine whether or not to execute a write access by first querying the file's state (e.g., file size or data).

their own observable events, replay them, and implement the appropriate switch–over actions for them.

## 6.6 Summary

The piecewise deterministic computation model is well suited for transparent replication of partitions. Replica consistency can be guaranteed by using a *semi–active replication* scheme based on view–synchronous group communication and forcing the follower replicas to follow the execution history observed on the leader. This is achieved by logging all non–deterministic events on the leader and informing the followers of this recorded event sequence before each *observable event*. The followers then replay events in the order logged.

Effects of failures are mostly local to the group; there is no rollback involved as with the approach based on a non–deterministic computation model described in the previous chapter. The only exception of this are failures of the leader replica after an observable event, but before the next synchronization of its followers. In this case, the new leader cannot tell whether the former leader failed before or after having executed the observable event, and it is therefore possible that the new leader executes this event a second time. For this reason, receiver–based *duplicate message detection* is mandatory. Duplicate message detection can be avoided only if observable events can be executed atomically with the synchronization of the followers.

In a heterogeneous distributed system, arbitrary partitions can be replicated only statically. Such statically replicated partitions are *k–resilient* objects: a partition replicated *k+1* times can tolerate up to *k* crash failures. Dynamic replication, i.e., the recovery or replacement of failed replicas at run time, is only possible in a heterogeneous system if the replicated partition fulfills a series of constraints such that all activity of the partition is triggered only by remote calls, and that these remote calls eventually terminate. Only under the *quiescence* assumption can new replicas be brought up to date with respect to the already running replicas. If all replicas of a partition execute on homogeneous physical nodes where system–level checkpoints including the tasking state can be taken, dynamic groups are possible without such restrictions.

*Replication transparency* (i.e., transparency towards other partitions in the system) 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, is also maintained, but only for k–resilient partitions. If recoveries are included in the model, state transfers necessitate the cooperation of the application level of the replicated partition.

# Chapter 7:

# Related Work

This section gives an overview of the state of the art in the domain of transactions in programming languages (i.e., outside the database–specific context) and of replication. I discuss some exemplary systems only, many others do of course exist. Clouds [DRAR91] and CHORUS [LJP93] for instance, two operating systems offering transactions as the prime structuring mechanism for processes, are not discussed because they're not directly in line with the integration of replication in a programming language.

The transactional systems comprise Argus, Camelot/Avalon, Arjuna, Isis, and Drago; Delta–4 and Manetho are two systems based on a piecewise deterministic computation model. Fault–tolerant Concurrent C is an early attempt to offer replication for fault tolerance for a general–purpose programming language that has certain similarities with Ada.

## 7.1 Circus

The Circus system [Coo85], developed 1984 – 85, is a system to transparently replicate program modules. A module in Cooper's terminology corresponds to an Ada 95 partition, but is assumed to behave deterministically. Replicas of a module form a troupe. The notion of replicated procedure calls was introduced to capture the many–to–many pattern of communication in RPCs between troupes. As troupes (and their members, the replicated modules) are deterministic, no coordination (and thus, no communication between troupe members) is necessary.

Initially, Circus did not allow any intra–module concurrency; only one RPC could be active at any time in a module. Later, the approach was extended to use transactions to handle concurrent RPCs; however, each RPC itself still had to be deterministic. At the time of publication of [Coo85], these transactions apparently were not implemented. In the paper, an extremely optimistic consistency protocol was proposed: replicas in a troupe would execute without any synchronization. When a transaction wanted to commit, the *clients* of the troupe would be notified, which then would deadlock if it was detected that any two troupe members tried to commit transactions in a different order. This deadlock would then have to be broken by aborting some transactions. It seems to me that this overly optimistic protocol would cause far too many aborts.

## 7.2 Argus

Argus is a programming language and run–time system to support the construction of fault–tolerant distributed programs. It was developed at MIT in the 80's (1981 – 88) [Lis88]. The programming language Argus is based on CLU.

A distributed application in Argus is structured as *guardians* [LS83] that encapsulate some state that is only accessible through RPCs. Remotely callable procedures are called *handlers*; they provide the only interface of guardians. A guardian's state is split into a volatile and a persistent state (atomic objects, [WL85]). Only the persistent state of a guardian can survive failures.

Remote calls in Argus are based on the concept of nested transactions [Mos81]. The Argus language offers constructs for application–level transactions, i.e., transactions are an integrated part of the language. Subtransactions can be executed concurrently using a special language construct (the `cobegin` statement). The transaction executing this `cobegin` statement is blocked until all the children have terminated; i.e., parent transactions cannot execute concurrently with their children. Guardians can handle requests concurrently, but Argus' tasking model is limited, and implemented within the Argus run–time support [LCJS87]. Deadlock detection is not provided.

Replication of guardians originally was not taken into account [Lis85]; failures were not masked. Implementing RPCs as nested transactions at least guarantees a consistent state of the persistent atomic objects. Operations on failed guardians must be re–issued by the client after recovery of the failed guardian (or the parent transaction might try something else, as it is informed of the abort of its child).

More recently, the Argus system has been augmented by transparent replication [Ghe90] of guardians using a primary–backup replication scheme based on viewstamped replication [Oki88].

## 7.3 Camelot and Avalon

Camelot [SE+87] is a facility for distributed transaction processing running on top of the Mach operating system. It was developed at Carnegie Mellon University in the late 80's (1985 – 92) as a successor of TABS [SD+85]. Avalon [EMS91] is a programming language for use with Camelot. It is heavily inspired by Argus, but extends a more widely used programming language: C++ instead of CLU. Avalon integrates nested transactions into C++; it follows the same restricted tasking model (`cobegin` statement) as Argus.

The main novelty of Camelot/Avalon is the use of a *hybrid atomicity* model [FLW92]: different objects in the system may employ different concurrency control schemes. An application designer can choose between locking or timestamp–based concurrency control.

## 7.4 Arjuna and Voltan

The Arjuna [SDP91] platform for reliable distributed systems was developed at the University of Newcastle upon Tyne, UK from 1985 on. Arjuna extends C++ by remote procedure calls and nested transactions. It uses its own preprocessor for RPC stub generation and runs on standard Unix. Arjuna is built upon the object–action model [SMR93].

Transactions are explicit; in fact, the Arjuna libraries make a lot of the internal classes visible (e.g., locks). It employs strict 2PL for concurrency control [PS88], although that could be changed by subclassing and overriding of methods.

Replication of objects is possible in Arjuna, albeit not transparently [PSWL95]: persistent objects must make explicit use of inherited operations for locking and state restoration. Although [LS90] describes a protocol for active replication of deterministic objects, Shrivastava states in [Shr94] that only passive object replication is allowed.

The Voltan system [SEST92], also developed at Newcastle, takes a different approach. It is based upon leader–follower replication, but currently only handles single threaded replicas [BLS98]. Limited non–determinism is handled (such as querying the clock, or non–determinism due to time–outs). Voltan is not transparent at all: it comes in the form of C++ libraries implementing the support for semi–active replication, but the application is responsible of using these libraries appropriately.

## 7.5 Isis and Horus

Isis [BR94] was the first group communication system based on the notion of view synchrony (or virtual synchrony). It offers the abstraction of a group together with operations for determining group membership and a set of multicast communication primitives with various semantics.

Isis is known in the first place for this model of group communication, but the Isis toolkit also includes a transaction manager [GBCR93] built on top of the group abstraction. Birman describes in [Bir85] the use of replication and transactions for building reliable distributed applications. Replicas are organized in a coordinator–cohort fashion; the coordinator may be different for each request. Although Birman never clearly describes his assumptions about the computational model, it can be deduced that replicas may be multithreaded, but that each transaction itself must behave deterministically, otherwise, the recovery protocol described in [Bir85] would not work correctly. Isis' transaction toolkit supports nested transactions; remote calls are viewed as nested transactions. This is not transparent, though: the application is offered e.g. constructs to declare new top–level transactions and to abort voluntarily.

Isis is the direct precursor of Horus [RBM96], a redesign for improving performance. Horus is very modular and improves Isis in several ways. It introduces a couple of new features, such as anonymous groups. ([GBCR93] mentions that using Isis groups to hold all the participants of a nested transaction tree leads to poor performance due to unnecessary accesses to a name server and a high rate of group joins and deletions.)

Both Isis and Horus are libraries; their use is not transparent if an application uses them directly.

## 7.6 Drago

Drago [MAAG96] is a recent (from 1994 on) language developed in Spain (Technical University of Madrid and University of Las Palmas de Gran Canaria). It has been designed and implemented as a fault–tolerant, distributed extension of Ada 83. Fault tolerance is achieved by (active) replication of virtual nodes, which are called *agents* in Drago and which are very similar to Ada's tasks. Like a task, agents have entries. The system ensures that all replicated agents accept entries in the same sequence.

Replicas in Drago form groups; all the replicated agents must be deterministic [GMAA97] (in particular, they must not contain local tasks). Drago also exploits groups as a naming abstraction by offering so–called cooperative groups whose member agents may be different and may behave non–deterministically. Replication is not transparent: a developer has to declare replicated agents as such in the source code.

Drago has been extended to include transactions [PJA98]. It follows more or less the traditional nested transaction approach except for allowing a parent transaction to execute concurrently with its children and for supporting multithreaded transactions. Transactions are a language construct (blocks encapsulated by `begin`/`end transaction` statements); the system uses locking for inter–transaction concurrency control, while concurrency control between the threads of a single transaction must be handled explicitly in the application using rendezvous.

The group concept is extended to cover transactional groups (both replicated and cooperative); transactional groups must be called from within a transaction and may themselves only call other transactional groups. Transactional agents handle requests from the same client serially, but requests from different clients are handled concurrently. The authors claim that replica consistency was guaranteed through deterministic scheduling. (But see the discussion of fig. 4.4 in section 4.4.2, page 50 of this thesis.)

## 7.7 replicAda

The replicAda system [HGC97], currently under development at the University Carlos III in Madrid, is an approach similar to mine to offer transparent replication of partitions for Ada 95. It is based on the same principles: a group communication toolkit is used; replicas of a partition form a group. The main difference is that they *assume* partitions to be deterministic, which they suppose is achieved by appropriate use of pragma `Restriction` (cf. the discussion in section 4.4.2 on page 50).

## 7.8 Fault–Tolerant Concurrent C

Fault–Tolerant Concurrent C (FTCC) [CGR88] is an extension of Concurrent C, which extends the C language with constructs for concurrency similar to those in Ada. Concurrent C has processes that communicate by rendezvous: processes have entries (just like Ada tasks), and there are `select` and `delay` statements similar to those of Ada. Processes can create other processes, but nesting of processes is not supported.

In FTCC, the programmer can explicitly create a process that is replicated. However, it is the programmer's responsibility that this semi–actively replicated process behaves deterministically. FTCC only guarantees that all replicas will make the same choices when a non–deterministic language construct is encountered (`select` or `delay`). This is a first step towards the piecewise deterministic model.

FTCC differs from Ada 95 in the choice of the unit of distribution and replication: in Ada 95, it is the partition, which may contain local tasks and which can handle multiple requests concurrently, whereas in FTCC, it is the process (task), which can handle requests only serially, as there is no nesting of processes.

## 7.9 Delta–4

Delta–4 [Pow91] was an ESPRIT project (1986 – 1992) on fault tolerance by replication. It provides an open architecture for developing fault–tolerant applications. It is very general in nature, offering several models based upon various failure assumptions [PCD90]. Replication schemes are classed as active [CPR+92], passive [SB89], or semi–active [BS93] replication. Delta–4 comes in two flavors: the Delta-4 OSA (Open Systems Architecture), and the Delta-4 XPA [BHV+90] (Extra Performance Architecture). The latter is targeted at the real–time world; it restricts the system to fail–silent components, and introduces semi–active replication.

Active replicas in Delta–4 must be deterministic (as usual). Passive and semi–active replication can be used for non–deterministic replicas, but then the assumption of fail–silent hosts must be guaranteed, e.g., by using a fail–silent network adapter (NAC). The Delta–4 architecture requires such NACs to be present in the system; they run some of the replica coordination protocols. NACs themselves always are fail–silent, whereas hosts may be subject to byzantine failures in active replication if the application has been configured to accommodate for it. Delta–4 assumes a synchronous network [Pow94].

## 7.10 Manetho

Developed at Rice University 1989 – 1993, Manetho [Eln93] is a system for transparent replication of processes in the V–System. It uses message logging [EZ94] and rollback recovery [EZ92a] using coordinated checkpointing. Client processes in Manetho are supposed to be piecewise deterministic; the system maintains an antecedence graph that describes the occurrence of non–deterministic events such as message receipts or scheduling decisions. Manetho makes heavy use of the features of the V–System to take and install checkpoints and also to optimize checkpointing. It also handles interactions of the application with the operating system kernel specially (system calls).

In [Eln93], Elnozahy states that replication is used for server availability; a leader–follower strategy is used [EZ92b]. The leader tracks non–deterministic internal events and delivers them like any other external event, allowing the followers to correctly replay them. Although described, it seems that the implementation does not support tracking of non–deterministic (internal) events [Eln93], limiting the approach to use replication only for deterministic S–components. Insofar, my work is the exact complement to his: my approach implements replication for fault tolerance of non–deterministic S– and CS–components, while Manetho can handle non–deterministic C–components.

## 7.11 Summary

The systems discussed can be classed in various ways. On the one hand, there are transactional systems (Argus, Camelot/Avalon, Arjuna, and Drago), and systems based on a piecewise deterministic model (Manetho). Some systems do not fit exactly in these two categories. Isis is a general group–communication toolkit that also includes a transaction manager; Delta–4 is a general architecture, but its XPA definition is based on a piecewise deterministic computation model. Fault–tolerant Concurrent C, replicAda, and Voltan all in some ways assume piecewise determinism, but handle only a subset of all possibly occurring non–deterministic events.

On the other hand, one can class these systems by their integration into programming languages. Argus, Avalon, and Drago are programming languages with integrated transactions. FTCC tries to incorporate replication in a programming language. Arjuna, Voltan, and Isis are toolkits in the form of libraries. Manetho and replicAda are attempts at transparent fault tolerance, in the case of Manetho based on message logging and coordinated checkpointing.

The transactional systems have proved useful, but they all operate in the context of programming languages that have a less rich semantics than Ada 95. The tasking (threading) facilities in these languages are restricted (Argus) or absent altogether (C++, where threads are created by means of system library calls). Tasking in Ada 95 makes it impossible to use this transactional approach in a transparent way due to the incompatibility of transactional concurrency control and the rich semantics of the features provided by Ada 95: application–defined scheduling control by means of barriers of protected objects may lead to unbreakable deadlocks that do not occur when RPCs do not have transactional semantics. Also, none of the transactional systems are transparent: transactions are an application–level construct for structuring computations.

The two systems based on a piecewise deterministic model (Manetho and Delta–4) are similar to my approach. Manetho is closely coupled to a particular operating system (V–System). Replication is supported by Manetho's implementation only for deterministic S– and CS–components. Delta–4 is a very general, elaborate framework for reliable distributed systems, and to a certain extent the approach described in chapter 6 and its implementation (RAPIDS, chapter 8) can be seen as an application of concepts of Delta–4's XPA to the specific case of Ada 95.

# Chapter 8:

# RAPIDS: An Implementation in Ada 95

This chapter discusses RAPIDS: "Replicated Ada Partitions In Distributed Systems", an implementation of semi–active replication as described in chapter 6 for the GNAT development system for Ada 95. I start with a general overview of this development system before detailing some of the finer points of the implementation.

## 8.1 Overview

The GNAT distribution includes an implementation of annex E of the Ada 95 language standard called GLADE [PT98]. It consists of two parts: an implementation of the run–time support for annex E — i.e., an implementation of the PCS —, which is called Garlic [KPT95], and a tool (`gnatdist`) allowing developers to configure their distributed applications. The compiler also takes part in this implementation of annex E as it generates the appropriate stubs for the remotely callable subprograms.

### 8.1.1 Building Distributed Applications with GLADE

Annex E of the Ada 95 language standard has been designed with the goal that the differences between centralized and distributed Ada 95 applications be as small as possible. Apart from the categorization pragmas with their associated restrictions and the fact that exceptions raised in asynchronously called remote subprograms are lost, the usual semantics of Ada 95 has been preserved in the distributed case.

Once a distributed application has been written, it must still be configured: library units must be assigned to partitions and partitions must be allocated on physical nodes. In the GNAT system, this configuration process is done off–line using a textual description written in a special–purpose configuration language [KNP96]. This configuration file is processed by `gnatdist`, which invokes the compiler to generate the appropriate stubs and links all the necessary run–time support routines to the library units in a partition, producing one executable per partition. It can also generate a starter application for the distributed application that automatically launches all partitions on their respective physical nodes. This attempt to give users the illusion of a centralized application can also be circumvented by suppressing starter generation and launching partitions manually.

### 8.1.2 The Structure of the Run–Time Support

The goal of transparent replication is largely met by implementing all replication–related activities within the run–time support of GNAT. Fig. 8.1 is a rough graphical representation of the structure of an Ada 95 partition in GNAT and shows the different components of this run–time support.

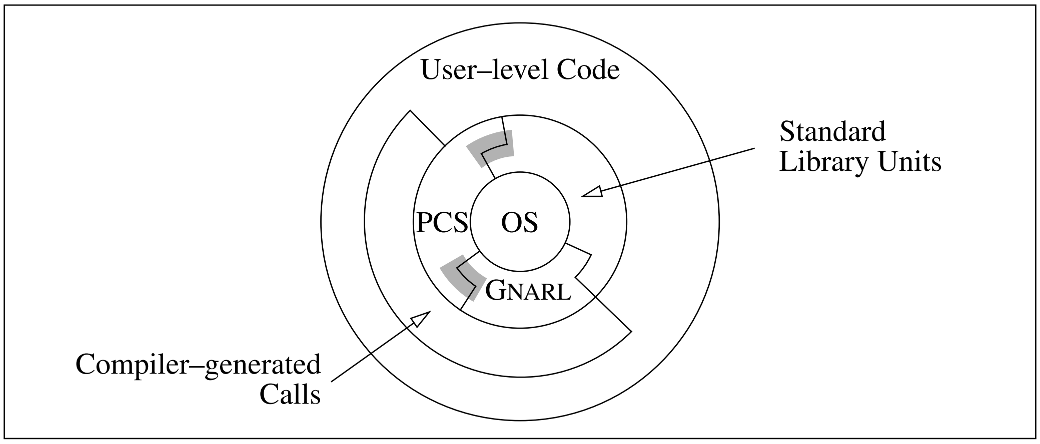


**Fig. 8.1:** Structure of a Partition

The run–time support consists of three main components:

- the PCS, whose implementation for GNAT is called Garlic,
- the tasking implementation, called GNARL (GNU Ada Run–time Library), and
- the implementations of all the standard library units.

The PCS itself depends upon the tasking implementation: Garlic executes RPC requests concurrently in separate tasks. It also uses some of the standard library units, for example `Ada.Unchecked_Deallocation`. Most of the standard library units do not directly use GNARL, except for the few standard packages dealing with tasks such as `Ada.Task_Information`. Both GNARL and the PCS are normally invisible for the application. The compiler automatically generates the necessary calls to GNARL for all tasking constructs of the language, and the PCS is usually only accessed by the compiler–generated stubs for remotely callable subprograms.

The shaded regions in fig. 8.1 correspond to the new replication manager: replication support permeates all three components of the run–time support. On the one hand, the replica manager has to be part of the PCS to be able to handle remote communications correctly; on the other hand, it must also be integrated with the tasking support to implement event replay of internal events. Finally, the replica manager offers an interface to its event log in a child package of `System.RPC`. This interface can be used by the implementation of the standard library units or by the application to define additional events and the necessary actions for replaying them.

### 8.1.3 A Short Tour of the PCS

The interface of the PCS is standardized in package `System.RPC`, discussed earlier in section 3.2.5. Fig. 8.2 shows the basic structure of GNAT's implementation of the PCS, called Garlic.

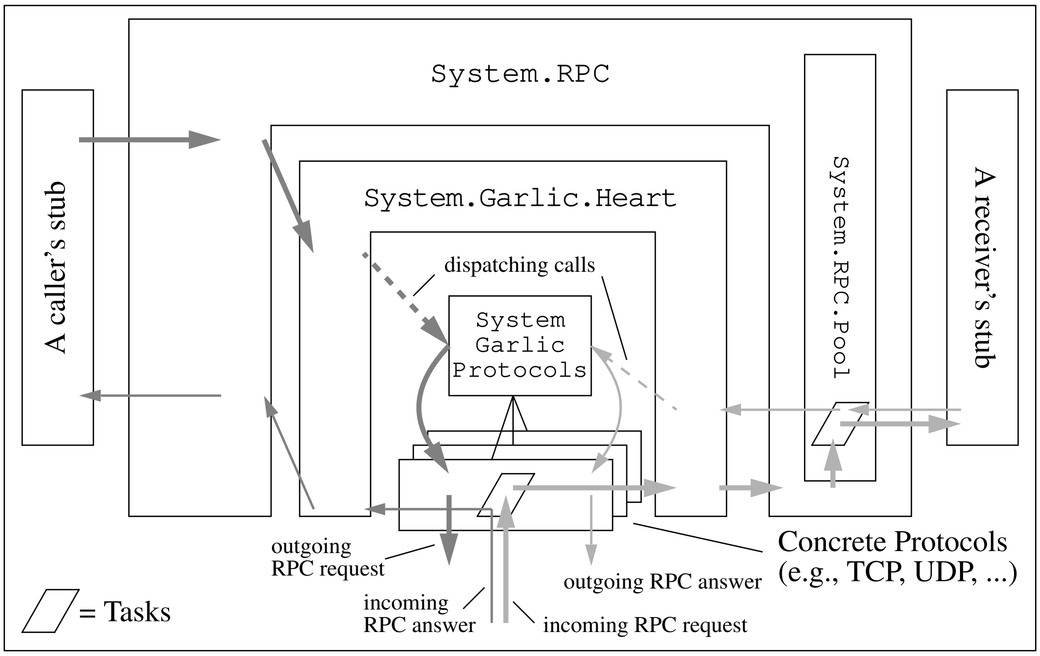


**Fig. 8.2:** Structure of the PCS

The standardized interface of `System.RPC` is based upon the implementation of package `System.Garlic.Heart`, which implements all the low–level message handling for remote subprogram calls. When a task makes a remote call, the caller's stub marshals the arguments and invokes `System.RPC.Do_RPC`, which in turn passes the stream containing the marshaled arguments and the `Partition_ID` of the destination partition to Garlic. Garlic then dynamically chooses the right message transmission protocol for the channel to that partition in `System.Garlic.Heart` and sends the RPC request. It also maintains the relationship between RPC request messages and the corresponding answers: the sending task enters the message sequence number of the request in a global list of pending requests and then blocks until the answer has arrived.

Physical protocols are all derived from an abstract tagged type declared in package `System.Garlic.Protocols`. Garlic invokes these protocols only through dispatching operations, it does not statically depend upon protocol implementations like `System.Garlic.TCP`. Upon its elaboration, each protocol registers itself with Garlic. Garlic also contains a table indicating which protocol is to be used on which channel. This table is generated by the configuration tool `gnatdist` according to the configuration file. The choice of the message transmission protocol can thus be made at configuration time, and only those protocols that actually are needed must be linked to the executable of a given partition.

Each protocol typically contains a task that waits for incoming messages. Whenever a message arrives, it simply forwards it to `System.Garlic.Heart`, which then checks the message format to determine whether it's an incoming request or an answer message. If it's an answer message, the task that sent the corresponding request in unblocked and given the message content for unmarshaling the remote call's results.

If an RPC request message arrives, an anonymous task is started in `System.RPC.Pool` to execute the remotely called subprogram. This task invokes the correct stub procedure to unmarshal the message content, execute the subprogram, and marshal the results. Once the stub has completed, the anonymous task passes the stream containing the marshaled results back to `System.Garlic.Heart`, which then sends the RPC answer message back to the calling partition[1].

### 8.1.4 The Tasking Implementation: An Overview of GNARL

GNARL, the tasking implementation for GNAT, has been designed to be maximally portable. System dependencies are encapsulated in a very low–level layer called GNULLI (GNU Low–Level Interface). The tasking semantics of Ada 95 are implemented using this interface, using whatever kinds of threads and locks the underlying operating system offers.

GNARL maintains a task control block (TCB) for each Ada 95 task, containing all the necessary information such as its stack size, its parent task, its priority, its entry queues, and so on. The TCBs of all tasks are linked together and stored in a global task list. Each task in Ada 95 has an associated `Task_ID`, which is implemented in GNAT as an access to the task's TCB: this uniquely identifies all tasks and gives GNARL direct access to its internal information given a simple `Task_ID`. The global task list is protected against concurrent accesses by *latches* — efficient, simple locks provided by GNULLI.

GNARL offers a procedural interface for use by the compiler. The compiler generates calls to GNARL's subprograms for each tasking construct in Ada 95. Fig. 8.3 shows a much

1. For efficiency reasons, `System.RPC.Pool` uses a pre–allocated pool of such anonymous tasks and re–uses them for successive remote calls. Dynamic task creations and destructions thus occur only if more RPCs are handled concurrently than there are tasks in this task pool.

simplified example that shows the basic principle (the true translation is much more complicated due to task termination, finalization, exceptions, and abort deferral). The `select` state-

```
task body T is

begin -- T
  loop
    select
      accept A do
        -- A's body
      end A;
    or
      accept B do
        -- B's body
      end B;
    or
      terminate;
    end select;
  end loop;
end T;
```

**Fig. 8.3a:** Ada Code for a Task

```
procedure T is

  procedure A is ...
  -- With A's body

  procedure B is ...
  -- With B's body

begin -- T
  loop
    declare
      Chosen_Entry : Entry_Index;
    begin
      Selective_Wait
        (Terminate_Mode,
         Chosen_Entry);
      case Chosen_Entry is
        when 1      => A;
        when 2      => B;
        when others => null;
      end case;
    end; -- block
  end loop;
end T;
```

**Fig. 8.3b:** Translated Code

ment is translated into a call to GNARL's `Selective_Wait` subprogram, which waits until an entry call occurs and then returns the index of the chosen entry. The compiled code then uses this index to execute the code corresponding to the entry.

All tasking–related language constructs are implemented ultimately by calls to GNARL. Task creation for instance is done by calling `Create_Task` with — amongst other data — an access to the task's code, i.e., to the procedure implementing it, as a parameter. Calling an entry also is translated into a call of a subprogram of the form `Task_Entry_Call (T'Task_ID, Entry_Index, ...)`, which takes care of queueing the entry call (making the caller wait) if that is necessary.

The compiler and GNARL also are closely intertwined in the implementation of protected objects. The data of a protected object is stored in a record that has some additional hidden fields for entry queues and a latch used to implement mutual exclusion. Procedure and function calls on a protected object call GNARL to lock and unlock the object's latch. Entry calls are also translated into calls to GNARL, which manages the entry queues and blocks and wakes up tasks as needed.

## 8.2 Global Structure of the Replication Manager

The replication manager is responsible for implementing semi–active replication. It implements both roles of replicas: on the leader replica, it builds the event log and transmits it at each observable event to the followers; on the follower replicas, it forces the partition to follow the execution history described by the event log and handles view changes. The replication manager is built around three main data structures:

- the event log,
- a list of system tasks for which no events are logged, and
- a status indicator telling whether the replica is the leader or a follower.

The event list is accessed from several places: the tasking support logs and replays internal events, the PCS logs and replays external events, and finally the standard library units (and the application) may log and replay additional events, e.g. system events such as accesses to `Ada.Calendar.Clock`. Fig. 8.3 shows the global structure of the replication manager.

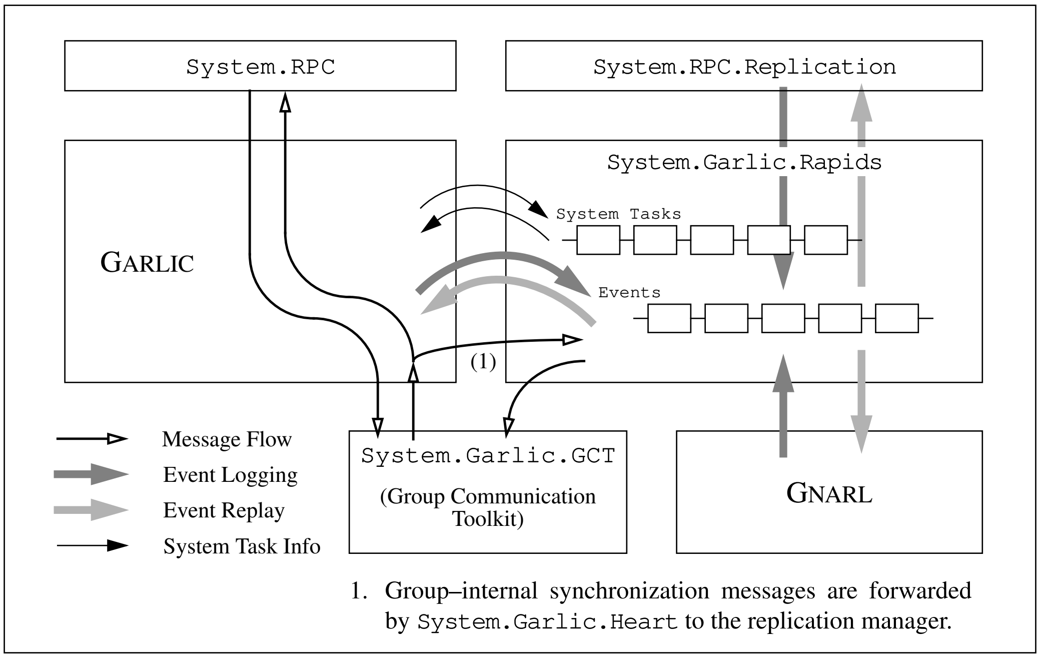


**Fig. 8.3:** Structure of the Replication Manager

The system task list contains the task IDs of all tasks for which the replication manager is not to do event logging and replay: these are all the tasks that are used within the PCS and the replication manager itself. If event replay were active for these tasks, the PCS and the replication manager would try to execute identically on all replicas, which must be avoided because the run–time support on the followers must do different things than on the leader.

There are three different interfaces for event logging and replay. The PCS uses direct calls to the replication manager. The standard library units use a public interface in a child package of `System.RPC`. (The Ada 95 language standard explicitly allow such an extension for providing additional interfaces, see [ISO95, E.5(26)].) Finally, internal events are logged and replayed by GNARL using callbacks to avoid making the tasking implementation depend statically on the replication manager.

## 8.3 The Group Communication Protocol

The group communication protocol assumed in the system model presented in section 4.2 is implemented as a new protocol that is added to Garlic. Any group communication protocol can be used for implementing semi–active replication provided it is based on view synchrony and offers reliable and reliable FIFO–ordered multicast. The current implementation is based on PHOENIX [MFSW95, Mal96], a view–synchronous group communication toolkit developed at the Operating Systems Lab at EPFL.

PHOENIX' model of computation includes a limited form of concurrency between requests based on the model of coroutines. However, managing concurrency is largely left to the application using PHOENIX. Since the coroutine model of PHOENIX does not integrate well with the tasking model implemented in GNARL, it was not possible to use the PHOENIX libraries directly within the Ada 95 run–time support. Instead, a small daemon application running on each physical node that executes a replica of a partition acts as a front end for PHOENIX. This is a separate application written in C++ (because the PHOENIX library offers a C++ interface) that provides a simple IPC interface to PHOENIX' group communication protocol. The Ada 95 group communication protocol only communicates with this C++ front end through IPC; the connection between the two is a bidirectional FIFO link. Because the Ada 95 partition and the daemon execute in different processes, Ada 95 tasks and the coroutines of PHOENIX do not interfere.

The interface to the group communication protocol is implemented like any other protocol as a derivation from the abstract protocol type in `System.Garlic.Protocols`. It offers the standard protocol interface with primitive operations to send messages. (Note that protocols have no primitive operation for receiving messages; they are active entities that forward received messages to `System.Garlic.Heart`.)

## 8.4 Events and Event Logging

Events are described by types derived from an abstract tagged root type `Event`, and the event log is implemented in `System.Garlic.Rapids` as a heterogeneous FIFO list that stores objects of the class–wide type `Event'Class`.

### 8.4.1 The Event Log

The public interface of the event log in `System.RPC.Replication` is shown in fig. 8.4 below.

```
package System.RPC.Replication is

  type Event is abstract tagged private;

  procedure Log (E : in Event'Class);
  -- Append a copy of the event to the event log, if the replica is
  -- the leader.

  procedure Get  (E    : in out Event'Class;
                  Valid:    out Boolean);
  -- Blocks the calling task until an event of the precise type (not
  -- the class!) of the actual parameter for the calling task is
  -- scheduled (Valid = True) or the log is exhausted and the replica
  -- now is a leader (Valid = False). The event remains in the log!

  procedure Remove;
  -- Removes the frontmost event from the log.

  procedure Send_Log;
  -- Sends the log to the followers and then empties it.

private

  type Task_GWID is ...;
  -- Group-wide task IDs, see section 8.5.

  No_Task : constant Task_GWID;

  type Event is abstract tagged
    record
      The_Task : Task_GWID := No_Task;
    end record;

  for Event'External_Tag use "System.RPC.Replication.Event";

end System.RPC.Replication;
```

**Fig. 8.4:** Event Logging Interface

When the `Log` subprogram is invoked, a copy of the event given as parameter is appended to the event log, provided the call occurs on the leader replica and the calling task is not one of the system tasks that are ignored. The component `The_Task` is set to the group–wide task ID (see section 8.5) of the calling task. If `Log` is called on a follower, or the calling task is a system task, the subprogram does nothing at all.

The `Get` subprogram is used to retrieve a logged event from the log. When it is called, it waits until the frontmost event in the event log is an event whose component `The_Task` matches the calling task's group–wide task ID and that has precisely the same type, i.e., the same external tag. It then sets the parameter `Is_Valid` to true and returns the event. If the event log contains no entries at all, the subprogram continues waiting until either the desired event finally appears (following a reception of an event log from the leader) or the partition becomes the new leader (after a failure of the former leader). If `Get` is called on a leader or the calling task is a system task, it only sets `Is_Valid` to false and returns immediately.

The `Remove` subprogram removes the frontmost entry in the event log, thereby unblocking one other waiting `Get` call.

The `Send_Log` subprogram is used to send intra–group synchronization messages. If called on the leader, it multicasts the event log to all replicas of the partition using the reliable FIFO–ordered multicast primitive provided by the group communication protocol and returns only when this synchronization message has been received. Only at that time the leader replica may continue — any time before it is not guaranteed that the followers could reproduce the observable event the leader is about to execute. On a follower, this subprogram has no effect.

To send the event log, `Send_Log` traverses the event log and marshals all events contained therein. Some care must be exercised when marshaling and unmarshaling the event log. The leader uses the class–wide stream attribute `Event'Class'Output` for all events in the event log for marshaling. The followers read in the event log using `Event'Class'Input`. These attributes use the external tag of the tagged type (written by the class–wide `'Output` attribute) to identify which derivation to read. However, the default value for this external tag is implementation–defined [ISO95, 13.3(75f)] — for all the standard says, it might well be derived from the type's internal tag, and there's no guarantee whatsoever that corresponding tagged types get the same internal tags on different replicas. It is therefore necessary to explicitly assign each type derived from the type `Event` an external tag using a representation clause of the form shown in fig. 8.4 above. Furthermore, events may contain components that have access types (like the `Checkpoint_Event` in fig. 8.16 on page 129): such events also must have explicitly defined stream attributes `'Write` and `'Read`.

### 8.4.2 Synchronizing the Replicas

The use of this interface to the event log is rather simple. Any operation corresponding to an event must be implemented following the pattern given in fig. 8.5 below. Note that there must not be any non–deterministic events involving the task executing `The_Operation` between the synchronization with the followers (`Send_Log`) and the execution of the event (in the statement `Do_Action` in the example). Also, using this pattern, the extended state interval recorded in the event log not necessarily starts with an observable event: because there may be several tasks active, it is quite possible that some other task logs events between the `Do_Action` statement and the logging of the event. This doesn't matter, though, because the `Send_Log` performed before `Do_Action` ensures that all followers will get to the point where `The_Operation` will perform the observable event `Do_Action`.

While it is not necessary to execute an event atomically with logging it, event replay on the followers must happen atomically. Consider as an example two tasks that call a procedure of a protected object. Obviously, the tasking support must *first* get the latch of the

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

  type Op_Event is new System.RPC.Replication.Event with
    record
      -- The characterizing data of the event.
    end record;

  for Op_Event'External_Tag use "An_Example.Op_Event";

  procedure The_Operation (...) is

    The_Event : Op_Event;
    Is_Valid  : Boolean;

  begin -- The_Operation
    System.RPC.Replication.Get (The_Event, Is_Valid);
    if Is_Valid then
      -- We're a follower, use 'The_Event' to perform the operation.
      System.RPC.Replication.Remove; -- Once done, remove the event.
    else -- We're the leader
      -- If it's an observable event, send the log
      System.RPC.Replication.Send_Log;
      -- Go ahead and execute the event
      Do_Action;
      -- Log all characterizing data of the event
      The_Event := (Event with ...);
      System.RPC.Replication.Log (The_Event);
    end if;
  end The_Operation;

end An_Example
```

**Fig. 8.5:** An Example Operation Causing an Event

protected object and *then* log the event (following the pattern in fig. 8.5). During replay, the follower must first wait for the event, then lock the protected object, and only then may it remove the event from the event list. If this sequence is not observed, there is no guarantee that the tasks actually are granted access to the protected object in the order logged.

This rule for handling event replay is a general one, it is not limited to the above example of tasks accessing a protected object. Event replay can be implemented safely by always following the sequence:

1. Wait for the event to be scheduled (frontmost in the log) using `Get`.
2. Execute the event.
3. `Remove` the frontmost event, which is the one gotten in step 1, from the log.

This allows atomic execution of an event during replay: other tasks remain blocked until the event is finally removed from the event log.

### 8.4.3 Interactions with GNARL

The event log, possibly being accessed concurrently from GNARL, the PCS, and the standard library units, must guarantee mutual exclusion for all operations. The accesses from GNARL in particular pose a few subtle problems in this respect.

To avoid having to fully integrate the tasking implementation with the PCS, which would be rather messy because then the PCS couldn't use the task concept anymore, I chose to have GNARL access the event log via upcalls. These callbacks are implemented by adding global variables of access–to–subprogram types to GNARL. By default, these variables are initialized to null, and GNARL makes an upcall only when they really refer to some routine. When `System.Garlic.Rapids` is elaborated, it installs access routines for its event log in them, and henceforth the tasking support will invoke them, thus logging (and replaying) events.

All subprograms in the tasking support that correspond to an internal event (e.g., task creation (`System.Tasking.Stages.Create_Task`), or selecting a call queued on a task entry (`System.Tasking.Rendezvous.Selective_Wait`), or making an entry call on a protected object (`System.Tasking.Protected_Objects.Protected_Entry_Call`)) have been rewritten to match the pattern shown in fig. 8.5, using upcalls to retrieve, log, or remove events. Placing these upcalls within GNARL must be done with great care. Because of possible concurrent accesses, the event log itself uses tasking constructs (and thus GNARL) to ensure mutual exclusion, i.e., upcalls from GNARL to the replication manager may result in recursive invocations of GNARL! The global tasking data structures within GNARL must therefore all be unlocked: GNARL must not hold any latch when an upcall is made, otherwise the partition may deadlock! It is not possible to temporarily release a latch in the tasking support when making an upcall as the deadlock freedom of GNARL itself relies on a strict policy defining the order in which latches must be locked and released. Luckily, it was possible to find in all cases a place for the upcalls where GNARL doesn't hold any latches anyway.

The event log in `System.Garlic.Rapids` cannot be implemented using protected objects to ensure task–safe manipulations. To break the aforementioned recursive invocation of GNARL, either GNARL or the callback routines must be able to recognize recursive accesses. The way protected objects are handled by the compiler and GNARL, they cannot be used for this purpose as individual protected objects cannot be identified within the run–time support. The event log is therefore encapsulated in a task, which can be identified at run time through its `Task_ID`. Whenever a callback is invoked with an event involving this event logging task, it does not access the event log but returns immediately, thereby terminating the recursion. Any rendezvous with this task is effectively ignored for event logging and replay.

The same technique is also used to encapsulate the list of system tasks, which are also ignored for event logging and replay. The tasks encapsulating the two lists must be handled specially, though. They cannot simply be treated like other system tasks: the whole scheme must be anchored somewhere!

## 8.5 Group–Wide Task Identification

Each event includes an identification of the task involved in the event (the component `The_Task` of type `Event` in fig. 8.4). The standard `Task_ID`s of Ada 95 only have a meaning within one single replica of a partition. If that partition is replicated, the replica manager has to make sure that the same tasks in different replicas get the same task ID.

Normally, task IDs in GNAT are implemented as pointers to the corresponding task control blocks (TCBs) within GNARL. If the partition is replicated, there is no guarantee that TCBs of corresponding tasks are allocated at the same address, and this scheme cannot be used for group–wide task IDs. An additional indirection is called for, mapping pointers to TCBs to integral values. The tasking support of GNARL has been changed by adding a new field in the TCB containing this group–wide task identifier. This is simply a unique integral value generated by the replication manager[1].

```
type Create_Task_Event is new Event with
  record
    Child : Task_GWID;
  end record;
```

**Fig. 8.6:** Task Creation Event

Whenever a task is created, the leader logs an internal event with the structure given in fig. 8.6. The type `Task_GWID` is the **g**roup–**w**ide task **id**entifier of the new task. When a follower creates a task, it allocates its internal data structures (e.g., the TCB) as usual. It then gets the event from the log it received from the leader and uses the `Task_GWID` found there for the new task.

Bootstrapping this process of assigning group–wide task IDs necessitates that all replicas agree on a common identification of the environment task. Because the precise value of the group–wide task identifier is of no importance, I have chosen an arbitrary value which is hardcoded in the replication manager: all replicas assign the environment task the value one as its group–wide identifier.

The standard type `Task_ID` remains unchanged. One cannot do very much with values of type `Task_ID`: except for the operations declared in `Ada.Task_Identification` [ISO95, C.7.1], `Ada.Task_Attributes` [ibid, C.7.2], `Ada.Dynamic_Priorities` [ibid, D.5], and `Ada.Asynchronous_Task_Control` [ibid, D.11], they can only be copied and compared. The implementation of the operations in the aforementioned packages must be changed, though: most of them must be implemented as internal events using event logging and

1. Note that one cannot use e.g. the numeric interpretation of some part of the TCB's address as allocated on the leader. After a failure, the new leader may allocate new TCBs for new tasks, and their group–wide IDs might coincide with old ones that were still generated on the former leader if they were derived from storage addresses. A numbering scheme independent of storage addressing must be used instead.

replaying, and `Ada.Task_Identification.Image` must be changed to use the task's group–wide ID to construct the string image of the `Task_ID` (which is allowed according to [ISO95, C.7.1(7)]: the string image is implementation–defined), otherwise any two replicas might evolve differently because the string images might be different.

## 8.6 Message Sequence Numbers

I have shown in section 6.4.4 that partitions (whether replicated or not) must be able to handle duplicate messages and in particular duplicate invocations gracefully. To this end, the PCS adds unique identifiers to all messages. A message sequence number is the combination of the partition ID of the sending partition and a monotonically increasing sequence number within that partition.

Ignoring repeated RPC answer messages is easy: if a leader receives an answer message (which is tagged with the sequence number of the corresponding request) for which it has no outstanding request, the answer is simply discarded. Followers discard answer messages only if the sequence number is smaller than or equal to that of the last stored answer message from that particular partition; otherwise, the message is stored for later use either during replay of the leader's event log or to avoid needlessly sending an RPC request message if the leader should fail.

Handling repeated invocations is similar. Each partition retains the results of a particular RPC request, i.e., the corresponding RPC answer message it sends back. If an RPC request arrives with a sequence number for which there is a retained result, this result from a prior invocation is returned. If a request that is still being handled arrives, the new request is discarded. The PCS maintains a list of the message sequence numbers of all currently active requests for this latter case.

Retaining results implies some sort of garbage collection, otherwise a partition would need to be able to store infinitely many answer messages. Garbage collection can also be implemented using the message sequence numbers. A partition basically needs a way to know which results have been received by the client. The simplest way to ensure this is to have the client to include in each request message it sends an expiration number: the highest message sequence number of stable requests. A request is *stable* if the result was received in an already completed extended state interval, and if all requests with lower message sequence numbers also are stable. All retained results for requests with message sequence numbers lower than this expiration number may be discarded.

The format of a message sent for an RPC thus becomes as shown in fig. 8.7. The PCS newly maintains a message sequence number for each partition. Formerly, it used only a single global counter for all messages sent, but this no longer suffices to implement expiration numbers, as garbage collection of retained results at one partition might be delayed if

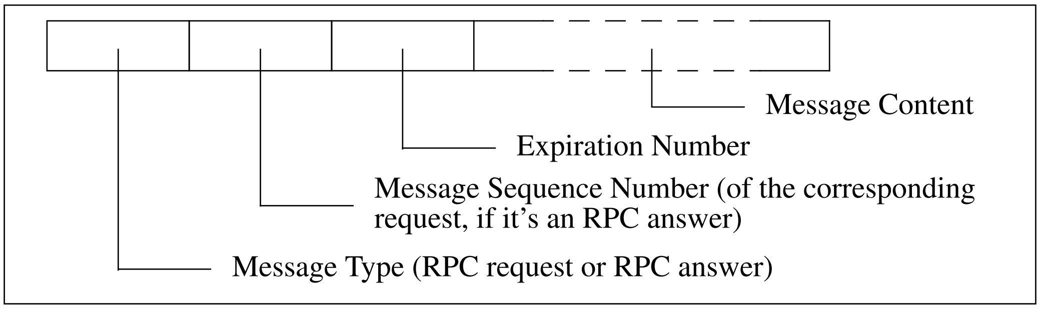


**Fig. 8.7:** Message Format

some other partition was slow. Expiration numbers also are maintained on a per–partition basis.

Assignments of message sequence numbers are internal events in order to guarantee that all replicas assign the same sequence numbers to corresponding messages. If this were not the case, duplicate messages could not be detected — the message sequence number is a system–wide unique identifier for the message, and therefore all replicas must choose the same ID for corresponding messages.

In practice, the domain of message sequence numbers is finite. At some point, a wrap–around from the highest sequence number to the lowest one occurs, and the receiver partition would basically have to be informed of this in order to be able to correctly garbage collect retained results. Superposing a protocol for this boundary case on the normal RPC protocol is complex. The main problem is that for some time after a wrap–around, a partition may have to deal with both old message sequence numbers generated before the wrap–around and new ones. Unfortunately, these new (low) values are indistinguishable from very old ones that also may be low.

I have adopted a pragmatic solution that avoids this difficulty altogether. Message sequence numbers are implemented as values in the range from 0 to $2^{64}$ (i.e., using 64bit integers). This range is large enough such that a wrap–around will not occur in any imaginable application[1].

## 8.7 Some Important Events

### 8.7.1 Internal Events

Internal events all concern tasking decisions. The common abstract ancestor type called `Event` (shown in fig. 8.4) has one component recording the group–wide task ID of the task that initiates the event. All other tasking–related events are derived from this root type. An example is the `Create_Task_Event` shown in fig. 8.6 above.

1. A quick back–of–the–envelope calculation shows that even if one million messages were sent per *second*, 64bit message sequence numbers will wrap around only after some 584'542 *years*!

Another example is shown in fig. 8.8: a `PO_Locked_Event` is added to the event log whenever a task obtains the latch of a protected object. It is used to log which task gains access to a protected object. Note that protected objects cannot be identified globally within the run–time support: they do not have unique IDs, like tasks do. Only logging the identity of the tasks (but not of the object they gain access to) suffices because each individual task executes sequentially. Any particular access to a protected object corresponds thus to exactly one `PO_Locked_Event`.

```
type PO_Locked_Event is
  new Event with
    null record;
```

**Fig. 8.8:** A `PO_Locked_Event`

The compiler translates each call of a function or procedure of a protected object by including calls to GNARL at the beginning and end of the subprogram to lock and unlock the protected object: an initial call to `System.Tasking.Protected_Objects.Lock`, and a closing call to `System.Tasking.Protected_Objects.Unlock`. The `Lock` subprogram, whose code is shown in fig. 8.9, waits until it can acquire the latch of the protected object and then gets it.

```
procedure Lock (Object : access Protection'Class) is
  -- A 'Protection' object contains the hidden data allocated for each
  -- protected object: the latch, entry queues, etc.

  Ceiling_Violation : Boolean;
  I_Am_A_Follower   : Boolean; -- 1

  function Task_ID_Self return Task_ID renames Tasking.Self;
  -- Gives access to the Task_ID of the calling task.

begin
  -- This is an upcall to the replication manager.
  Call_PO_Lock_Hook (The_Task => Task_ID_Self,
                     Is_Valid => I_Am_A_Follower);
  -- 'Object.L' is the latch of the protected object.
  Write_Lock (Object.L'Access, Ceiling_Violation);
  if I_Am_A_Follower then
    Call_Remove_Hook; -- Upcall
  else
    Call_PO_Locked_Hook (The_Task => Task_ID_Self); -- Upcall
  end if;
  if Ceiling_Violation then
    Raise_Exception (Program_Error'Identity, "Ceiling Violation");
  end if;
end Lock;
```

1. The shaded regions show the code added for replication management

**Fig. 8.9:** Locking a Protected Object

The callback to the replication manager invoked by `Call_PO_Lock_Hook` ultimately calls the `Get` access routine of the event log, with an event of type `PO_Locked_Event` as argument. On a leader, this is a no–op, and returns `False` in `Is_Valid`. The leader then goes on and tries to acquire the latch of the protected object through `Write_Lock`. Once it has got the latch, it logs a PO_Locked_Event with the Task_ID of the calling task through the callback Call_PO_Locked_Hook. On a follower, the Get routine invoked from Call_PO_Lock_Hook waits until an event of type PO_Locked_Event with a group–wide task ID matching that of the given task is first in the event list. It then returns that event and sets Is_Valid to True. The follower then also tries to acquire the latch: barring a potential ceiling violation, this will always succeed immediately because any other task that might try to access the same protected object is still blocked in its call to Call_PO_Lock_Hook. Once it has got the lock, it removes the event from the event list using the callback Call_Remove_Hook.

Other internal events are implemented in GNARL in the same spirit. There are callbacks and corresponding event types for all the tasking events identified in section 6.4.1, and the subprograms in GNARL implementing these events have been changed accordingly by inserting callbacks for event logging and replay. System events, i.e., non–deterministic events not directly linked to tasking, also are handled in this way.

### 8.7.2 Clock Events

```
type Clock_Event is new Event with
  record
    Year, Month, Day : Integer;
    Seconds          : Duration;
  end record;
```

**Fig. 8.10:** A Clock_Event

Events describing querying the clock, e.g. through Ada.Calendar.Clock, are handled specially. A Clock_Event as shown in fig. 8.10 is used for event logging. The event contains the time value the event produced on the leader, and followers will return this value instead of their own current time. In addition, followers also store the difference of the leader's time and their own time. After a failure of the leader, the new leader will correct all subsequent accesses to its clock by this difference. In this way, it is made sure that time never runs backwards for the application: the whole group logically always runs on the initial leader's time, even after a failure of that initial leader.

This also applies to absolute delay statements (delay until): their expiration time is equally adjusted. Because delay expirations also are internal events, it doesn't actually matter for how long exactly a task containing a delay statement is delayed: the outcome will be replayed according to the event log; only the logical time order is respected. All delay statements are translated into absolute delays within GNARL and implemented in a subprogram Timed_Delay. This subprogram newly has a structure corresponding to the pseudo–code shown in fig. 8.11.

As shown, a delay statement generates two events: one for its start, and a second one when it expires. The Call_Delay_Start_Hook callback ultimately invokes the Get routine to get the starting event. On the leader, it returns immediately (leaving Expiration unmodified). The leader then logs the expiration time and maps it — as it is expressed in the initial leader's time frame — to its own local time frame before delaying and finally logging the expiration event.

```
procedure Timed_Delay
   (Self : in Task_ID;
    T    : in Time) is

   Expiration      : Time := T;
   I_Am_A_Follower : Boolean;

begin -- Timed_Delay
   Call_Delay_Start_Hook
     (Self, Expiration,
      I_Am_A_Follower);
   if not I_Am_A_Follower then
      -- Leader
      Call_Delay_Start_Log_Hook
        (Self, Expiration);
      Map_To_Local_Time (Expiration);
      OS_Delay_Until (Expiration);
      Call_Delay_Expiration_Hook
        (Self);
   else
      -- Follower
      Call_Remove_Hook;
      Call_Delay_Expired_Hook
        (Self, I_Am_A_Follower);
      if not I_Am_A_Follower then
         -- This follower has meanwhile
         -- become the leader!
         Map_To_Local_Time
           (Expiration);
         OS_Delay_Until (Expiration);
         Call_Delay_Expiration_Hook
           (Self);
      else -- Still a follower...
         Call_Remove_Hook;
      end if; -- Has become leader?
   end if; -- Is leader?
end Timed_Delay;
```

**Fig. 8.11:** Logging and Replay of Delays

On a follower, `Call_Delay_Start_Hook` waits for the starting event to be scheduled before returning. When it *does* return, `Expiration` contains the expiration time on the leader. The follower removes this event from the log and waits for the expiration event to be scheduled by calling `Call_Delay_Expired_Hook`. When that upcall returns, there are two possibilities:

- The follower has become the leader of the group because the former leader has failed in the meantime. The moment the replica becomes the new leader, `Call_Delay_Expired_Hook` returns with `I_Am_A_Follower` set to `False`. In this case, the new leader (and former follower) must actually delay until the expiration time (after converting it to its own time frame) and then log an expiration event.
- The replica is still a follower, and the `Call_Delay_Expired_Hook` upcall returns with `I_Am_A_Follower` set to `True` because the event is scheduled. In this case, it suffices to remove that event.

### 8.7.3 External Events

External events are special inasmuch as followers do not replay them. Followers do not physically send RPC requests to other partitions; they only behave as if they did and wait for an answer to arrive. It wouldn't actually hurt to send RPC requests from a follower because the duplicate message detection at the receiver's side would take care of such redundant requests, but assuming that all communication with a group of replicas is by reliable multicast, it isn't necessary. RPC answer messages also are not sent by followers: the leader is the only replica that interacts with the system beyond the group of replicas.

Events corresponding to incoming RPC messages identify the appropriate messages only through their message sequence numbers. Since all messages to the group are broadcast reliably, it is guaranteed that all replicas have received or will receive them. It is not necessary to include the message itself in the events logged!

Sending an RPC request message is implemented according to the code outlined in fig. 8.12.

```
type Send_Status is
  (Success, Failure);

type RPC_Call_Event is
  new Event with
    record
      Msg_ID : Sequence_Number;
      Result : Status;
    end record;
...
procedure Send_Request
  (Dest : in     Partition_ID;
   By   : in     Protocol_Ref;
   Id   : in     Sequence_Number;
   Msg  : access Param_Strem_Type)
is

  Send_Event    : RPC_Call_Event;
  Status        : Send_Status;
  Seq_Num       : Sequence_Number
    := Id;
  Im_A_Follower : Boolean := False;

begin -- Send_Request
  Get (Send_Event, Im_A_Follower);
  if not Im_A_Follower then
    Send_Log;
    -- 'Send' is dispatching!
    Send
      (By, Dest, Id, Msg, Status);
    -- Update internal data
    -- structures as needed
    Send_Event :=
      (Event with Msg_ID => Id;
                  Result => Status);
    Log (Send_Event);
  else
    -- Follower: update data
    -- structures as if we had sent
    -- the msg.
    Remove;
    Status  := Send_Event.Result;
    Seq_Num := Send_Event.Msg_ID;
  end if; -- Is leader?
  if Status = Failure then
    raise Communication_Error;
  end if; -- Sending worked OK?
  ...
end Send_Request;
```

**Fig. 8.12:** Handling of an External Event

Note that external events must be logged even though followers do not replay them. Logging observable events, i.e., events corresponding to sending a request or an answer message, allows followers to avoid re–executing them during replay, and logging events corresponding to message receptions allows replicas to delay handling of incoming requests while taking checkpoints (see section 8.10).

## 8.8 Remote Access Types

Special care must be taken to handle distributed objects in a replicated partition correctly. The standard's basic model of remote access types is that of a *fat pointer*: a remote access value is a tuple containing both the local address of the distributed object within the partition it resides on and the `Partition_ID` of the said partition. Remote accesses can only be stored, compared, passed around from one partition to another, and dereferenced in the context of a remote dispatching call. To do a remote dispatching call, the caller uses the Partition_ID part of the remote access value to identify the destination partition to which it then sends the request for remote dispatching. The destination partition then uses the local address part of the remote access value to identify at run–time the object that is referenced.

This simple scheme does not work reliably when partitions are replicated. Even in a homogeneous distributed system, there is no guarantee that different replicas allocate corresponding objects at the same addresses. There are two issues that need to be addressed to handle remote access types:

- *Heterogeneity*: how can remote access be implemented in a heterogeneous system, where different partitions might use different representations of remote access values?
- *Replication*: how can they be implemented such that all replicas agree upon the object referred to by a particular remote access value?

The solutions to both problems are translation tables. The basic principle is that each partition and each replica of a partition maps its local representation to a common external representation for any given remote access value.

Remote access values in heterogeneous distributed systems can be implemented using a translation at the partitions using them. A partition P can obtain a remote access to an object residing on some other partition Q only as the result of a remote subprogram call to Q. What P actually receives is of course the marshaled representation of the remote access value. In the classical "fat pointer" model, P then unmarshals the value and subsequently uses this unmarshaled value.

In a heterogeneous distributed system, this approach may fail because P's representation of an access value may not be able to store an access value local to partition Q. Consider a case where a partition Q uses the pair <Partition_ID, 64bit local address> as a remote access value, while another partition P uses the representation <Partition_ID, 32bit local address>. Clearly, P's representation cannot hold an unmarshaled remote access from Q: the 32bit local address cannot store a 64bit address local to Q.

The solution is a simple hashing scheme. Instead of using the unmarshaled value of the remote access value, P uses the *marshaled* version. This is feasible because no operations that would depend on the precise value of the "local address" part are possible with a remote access value. Whenever a partition receives a marshaled remote access, it stores this marshaled representation in some data structure. Internally, it uses a local representative for this stored marshaled value; this representative contains a reference to the stored marshaled value. Partitions only have to have a common way to extract the Partition_ID part of the remote access value; the precise format of the "local address" part may remain unknown as the only partition that will ever need to use it is the one that generated it: the one the object resides on. Whenever a remote access value is passed on from one partition to another, the stored original marshaled representation is actually used.

```
procedure Marshal_Remote_Access
   (Stream : access Root_Stream_Type;
    Item   : in     Remote_Access)
is
   My_ID : Partition_ID := ...
   -- Get ID of this partition from
   -- the PCS.
begin --  Marshal_Remote_Access
   Partition_ID'Write
      (Stream, Item.ID);
   if Item.ID = My_ID then
      -- It's a remote access to some
      -- object on this partition:
      -- marshal 'Item'.
      System.Address'Write
         (Stream, Item.Addr);
   else
      -- It's a remote access to some
      -- other partition: use stored
      -- value!
      Append_To_Stream
         (Stream,
          Stored_Value (Item.Addr));
   end if;
end Marshal_Remote_Access;
```

```
function Unmarshal_Remote_Access
   (Stream : access Root_Stream_Type)
   return Remote_Access
is
   My_ID       : Partition_ID := ...
   -- Get ID of this partition from
   -- the PCS.
   Local_Value : Remote_Access;
begin --  Unmarshal_Remote_Access
   Partition_ID'Read
      (Stream, Local_Value.ID);
   if Local_Value.ID = My_ID then
      -- It's a remote access to an
      -- object in this partition:
      -- really unmarshal it!
      System.Address'Read
         (Stream, Local_Value.Addr);
   else
      -- It's a remote access to an
      -- object on some other parti-
      -- tion: store marshaled version
      -- in a global data structure,
      -- return a handle to the stored
      -- value.
      Local_Value.Addr :=
         Store_Value (Stream);
   end if;
   return Local_Value;
end Unmarshal_Remote_Access;
```

**Fig. 8.13:** Marshaling and Unmarshaling Remote Access Types

This change in the handling of remote access values can be implemented transparently in the implementation of the stream attributes used for marshaling and unmarshaling them. In GNAT, the default stream attributes for all standard types are defined in a package called `System.Stream_Attributes`. For each occurrence of a stream attribute (`'Read`, `'Write`, `'Input`, or `'Output`) of a standard type, the compiler actually generates a call to a subprogram of this package. Changing the implementation of the marshaling and unmarshaling routines for remote access types according to the pseudo–code given in fig. 8.13 automatically implements the proposed hashing scheme for all remote access types[1].

1. In the interest of clarity, I have slightly oversimplified the pseudo–code in figures fig. 8.13. Of course, one does not store the whole remaining `Stream` in `Store_Value` but only that portion that corresponds to the marshaled remote access value (including the `Partition_ID`). During marshaling, one marshals the local address part in a `Stream_Element_Array` and uses `'Output` to append *that* to the stream instead of the `System.Address'Write` shown above. An analogous operation is then needed instead of `System.Address'Read` when unmarshaling. A similar caveat must be made for fig. 8.14 below: not the whole remaining stream is logged and put in the translation table but only the relevant part. Because these manipulations are rather tricky and only obscure the pseudo–code, I have left them out.

In order to avoid memory leaks, the local representatives for remote access values should be implemented as controlled types. If the type `Remote_Access` itself is derived from `Ada.Finalization.Controlled`, reference counting can be used to discard the stored marshaled representation allocated by `Store_Value` once there's no longer any local representative referencing it.

In this way, partitions on heterogeneous nodes can work with remote access values despite having different internal representations for remote access types, in particular their "local address" part. The only partition that ever really unmarshals a local address is the one that originally marshaled it; all other partitions only copy the original marshaled representation.

When a partition is replicated, a second problem comes up: the replication manager must make sure that all replicas agree upon which remote access value refers to which object. If they did not agree, a failure of the leader might result in erroneous execution. The new leader must be able to handle remote access values that were generated by its predecessor. The solution for this case is similar to the one described above, but this time a translation table is used on the replicas for remote access values that refer to objects on the replicated partition. (In the above case, translation is done for remote access values referring to objects on *other* partitions.)

The leader replica marshals remote accesses to its objects as described above. In addition, it also logs this marshaling as an internal event in the event log. The log entry contains the marshaled representation of the remote access. Whenever a follower encounters such a log entry during replay, it adds the pair `<Leader's marshaled remote access, Follower's local value of the remote access>` to a translation table. When the follower becomes the new leader, it always checks this table upon unmarshaling a remote access: if it receives a marshaled remote access that is in the table, it uses the corresponding local entry in the table as the unmarshaled value. It also checks the table whenever it is about to marshal a remote access value referencing an object on the partition: if an entry is found, it uses the former leader's marshaled representation! In this way, other partitions always get unique and stable, unchanging remote access values for one particular distributed object, and therefore comparisons of remote access values and remote dispatching calls continue to work even if a leader replica fails. The pseudo–code for the marshaling and unmarshaling operations (`'Write` and `'Read`) is shown in fig. 8.14.[1]

Note that there are now two different tables involved! For remote access values referencing objects that reside on other partitions, there's a table that stores the original marshaled representation of the remote access and returns a local handle (address) for it. This translation is not necessary for an application executing on a homogeneous distributed system. A second table is used on replicas to translate the leader's marshaled remote access values into local equivalents on the followers, guaranteeing that all replicas use the same

1. Note that the pattern given in fig. 8.5 reappears in the central part of `Marshal_Remote_Access`.

```
procedure Marshal_Remote_Access  (Stream : access Root_Stream_Type;
                                  Item   : in     Remote_Access)
is
  My_ID : Partition_ID := ...
  -- Get ID of this partition from the PCS.
begin -- Marshal_Remote_Access
  if Item.ID = My_ID then
    -- 'Item' is a remote access to some object on this partition.
    -- Check the translation table to see if there's already a value.
    declare
      S : Translation_Ref := Translated_Value (Item);
    begin
      if S /= null then
        Append_To_Stream (Stream, S.Marshaled_Value);
      else -- No, this value is a new one!
        declare
          E             : Marshaling_Event;
          Im_A_Follower : Boolean;
        begin
          Replication.Get (E, Im_A_Follower);
          if Im_A_Follower then -- We're a follower!
            -- Add entry to translation table, using the value from the
            -- logged event.
            Add_Translation (E.Marshaled_Value, Item);
            Append_To_Stream (Stream, E.Marshaled_Value);
            Replication.Remove;
          else -- We're the leader!
            -- Marshal item; don't use Item'Write: recursive call!
            Partition_ID'Write (Stream, Item.ID);
            System.Address'Write (Stream, Item.Addr);
            Add_Translation (Stream, Item);
            -- Log event
            E := (Event with Marshaled_Value => Stream);
            Replication.Log (E);
          end if;
        end; -- block
      end if; -- Value found in translation table?
    end; -- block
  else
    -- It's a remote access to some other partition: use stored value!
    Append_To_Stream (Stream, Stored_Value (Item.Addr));
  end if;
end Marshal_Remote_Access;
```

**Fig. 8.14:** Marshaling and Unmarshaling of Remote Access Types on Replicas

marshaled values for accesses to replicas of the same object. This second translation is always needed when a partition is replicated, even in homogeneous distributed systems.

A replica should remove entries in this translation table whenever the referenced object ceases to exist, otherwise there is a possible memory leak and the translation table will progressively grow to unbounded size. The compiler should therefore implement dis-

```
function Unmarshal_Remote_Access (Stream : access Root_Stream_Type)
  return Remote_Access
is
  My_ID       : Partition_ID := ...
  --  Get ID of this partition from the PCS.
  Local_Value : Remote_Access;

begin --  Unmarshal_Remote_Access
  Partition_ID'Read (Stream, Local_Value.ID);
  if Local_Value.ID = My_ID then
    -- 'Item' is a remote access to an object in this partition.
    -- Check the translation table.
    declare
      S : Translation_Ref := Translated_Value (Stream);
    begin
      if S /= null then
        Local_Value.Addr := S.Local_Value;
      else
        -- Nothing found: the "local address" part has indeed been
        -- generated on this replica! Really unmarshal it!
        System.Address'Read (Stream, Local_Value.Addr);
      end if;
    end; -- block
  else
    -- It's a remote access to an object on some other partition: store
    -- marshaled version in a global data structure, return a handle to
    -- the stored value.
    Local_Value.Addr := Store_Value (Stream);
  end if;
  return Local_Value;
end Unmarshal_Remote_Access;
```

**Fig. 8.14 (continued):** Marshaling and Unmarshaling of Remote Access Types...

tributed objects as having a hidden component of a controlled type whose `Finalize` operation deallocates the object's entry in the translation table if one exists[1].

## 8.9 Failures

A failure of a replica in the group changes the group's composition and thus provokes a view change. PHOENIX delivers a view change message containing the new group composition to all members in the new view.

The leader replica can silently ignore these messages — if it has not failed, it continues to be the leader. A follower, however, may have to take action upon receiving a view change message: if the leader fails, one of the followers must become the new leader. The leader election protocol in this case is very simple and requires no extra communication: the

1. Note that the Ada Rapporteur Group decreed in AI–126 that `Ada.Finalization` *is* a `Remote_Types` package.

group composition in PHOENIX' view change message is ordered, and the first replica in this list becomes the new leader.

The new leader first continues replaying whatever events it still has in its event log to bring itself up to date with the last known state of the former leader. Messages arriving during this replay are buffered until the event list has been fully replayed. Only then the replication manager switches to leader mode, unblocking the delayed message deliveries. Subsequently, the replica assumes the role of the leader of the group.

## 8.10 State Transfers

When a new replica joins the group to replace a failed one, a view change with an ensuing state transfer occurs: the new replica receives the current state from one of the group members. This sub–section discusses the implementation of state transfers in RAPIDS.

### 8.10.1 Collecting the State

Given the possibility of a heterogeneous distributed system, state transfers are implemented based on quiescence as discussed in section 6.4.5. It is assumed that

- replicated partitions are only S– or CS–components, where all activity basically is triggered by incoming RPC requests, and
- there are quiescent moments in the execution of a partition where no remote calls are active.

The full list of conditions has been given in section 6.4.5 on page 91. Under these assumptions, a partition is quiescent when the aforementioned list of currently active requests is empty. An anonymous task that is to handle an incoming remote call request registers itself and the request message with the replication manager and unregisters when it is done handling the request, i.e., after it has sent back the corresponding answer message or, if the remote call was asynchronous, when the remotely called subprogram has terminated. Taking a checkpoint consists of the following steps:

- Event logging is blocked. This makes all tasks that try to log an event block. The only tasks that — by the quiescence assumption — could be active and try to log events are system tasks within the PCS itself. They could try to log `RPC_Request_Received` events for RPC requests that arrived since quiescence was detected. By blocking them, they are effectively delayed until the checkpoint has been taken.

- A system task is started that calls the `Get_State` subprogram, which the application installed as a callback routine in the run–time support, using the interface in package `System.RPC.Replication` shown in fig. 8.15. If no `Get_State` subprogram is installed, the replication manager obviously does not even try to take a checkpoint.

```
package System.RPC.Replication is
  ...
  type State_Collector is access
    procedure
     (State : access Param_Stream_Type);
  procedure Install
    (Collect : in State_Collector);
  ...
end System.RPC.Replication;
```

**Fig. 8.15:** The `Get_State` Callback

- The `Get_State` subprogram collects the application's state, marshaling it into the given `Param_Stream_Type`. When `Get_State` has terminated, the system task collects all relevant state within the PCS and adds it to that stream.
- It then atomically adds a checkpoint event containing the collected state as shown in fig. 8.16 to the event list and unblocks event logging, allowing normal processing to continue.

```
type Checkpoint_Event is new Event with
  record
    The_State : Param_Stream_Access;
  end record;
```

**Fig. 8.16:** A `Checkpoint_Event`

When a view change occurs, newly joining replicas get the last checkpoint plus all the logged events since then as their initial state. This means that `System.Garlic.Rapids` maintains not one event list but two:

- The first one is used for normal synchronization and has been discussed in section 8.4.1 above. It is built on the leader and purged whenever the list's contents are sent to the followers at an observable event. It therefore contains at most one extended state interval. On the followers, this list is emptied successively during replay.
- The second one may cover a longer interval than the first one: it contains all events that occurred since the last checkpoint and may comprise several extended state intervals. It always starts with a checkpoint event, and it is purged whenever a new checkpoint is being taken. This second event log is called the *shadow log*.

Whenever an event is removed from the first list, it is actually appended to the second one. On the leader, this happens when a synchronization message is sent; on a follower, this occurs each time an event is removed after having been replayed.

Followers do not replay checkpoint events. As the event already contains the marshaled state as it had been collected on the leader, calling `Get_State` on a follower would be simply superfluous. Whenever a checkpoint event is the first event in a follower's event log, it purges its shadow event log and then puts the checkpoint event onto the shadow log. Subsequent events are added to the shadow log whenever they are replayed.

### 8.10.2 Transferring the State

When a view change from view $\mathbf{V_i}$ to a new view $\mathbf{V_{i+1}}$ occurs and a state transfer is necessary (i.e., some of the replicas in $\mathbf{V_{i+1}}$ joined the group), the PHOENIX group communication protocol guarantees the following:

- On a group member that also was in view $\mathbf{V_i}$, PHOENIX invokes a callback `get_state` to get the current state of the group. This callback occurs after delivery of the view change, but before any message for view $\mathbf{V_{i+1}}$ is delivered.
- On a newly joining replica, a `put_state` callback is invoked after the new view has been delivered, but before any message delivery for this view. The `put_state` callback passes the state collected by `get_state` to the new replica.

Both these callbacks are implemented in the C++ wrapper daemon by sending a message through IPC to the connected Ada 95 partition and waiting for a reply. At the PCS, `get_state` and `put_state` thus arrive as normal messages, but the order guaranteed by PHOENIX is maintained because of the FIFO link between the daemon and the partition.

When the leader gets a `get_state` message, it returns its shadow event log to PHOENIX. If the `get_state` message happens to arrive while the state is being collected by the application–level `Get_State` subprogram invoked when quiescence is detected, the replica manager first lets `Get_State` terminate. When a follower receives this message, it first continues replaying events until its main event log is exhausted. Only then does it return its shadow event log to PHOENIX. The state PHOENIX gets is thus always the same, regardless of the replica that collected the state.

Note that the leader does *not* have to interrupt execution of requests that are active when the `get_state` message arrives and is handled. The only events that could possibly interfere with a state transfer would be those that also caused accesses to the shadow event log: observable events. Observable events cause the event log to be sent to the followers in a synchronization message. The coherence of the shadow log is guaranteed if `Send_Log` first sends the log[1], then waits until this synchronization message is delivered, and only then moves the events to the shadow log. This works because a multicast in PHOENIX is also delivered on the sender, and because PHOENIX delivers messages only once the state transfer via `get_state` and `put_state` has been done. Activities of requests being handled during the state transfer will therefore be sent to all followers — including the replicas that have newly joined — in the next synchronization message.

### 8.10.3 Installing the State

A replica that joins a group always assumes the role of a follower. (Unless it is the first one to join the group, which is thereby created: this replica is the initial leader. But this special

1. Using a reliable FIFO–ordered multicast, cf. section 8.4.1.

bootstrapping case is not of interest here because there is no state transfer involved.) The first message it delivers after having joined the group is a view change, followed by a `put_state` operation with the stream obtained via `get_state` on one of the old group members. It installs this state by making an upcall to an application–defined subprogram `Put_State`, which is defined and installed within the replication manager in the same way (using an access–to–subprogram) as the `Get_State` subprogram. If the application didn't yet install such a `Put_State` callback, the replication manager waits until this happens. In this way, the application can first complete its elaboration before a state is to be installed.

## 8.11 The Configuration Language

```
Partition_Declaration :=
    Defining_Identifier_List ":"
    ["replicated"] "Partition"
    [":=" Enumeration_Of_Ada_Units].1
```

1. See [KNP96] for an overview of the full syntax of `gnatdist`'s configuration language.

**Fig. 8.17:** New Configuration Language Syntax

Currently, there is only one parameter that can be specified in the configuration language: the fact that a partition is replicated, and on which physical nodes these replicas shall be allocated to. The syntax of the configuration language [KNP96] has been slightly changed by adding a new keyword `replicated` to the definition of a partition as shown in fig. 8.17. The `Defining_Identifier_List` declares a number of partitions, possibly initialized by assigning some Ada units to them. The keyword `replicated` now indicates that these partitions are replicated, enforcing a couple of constraints on the further definitions for these partitions:

- Starter generation must be suppressed. This restriction is planned to be removed in the future, but for the time being, partitions must be launched manually.
- There must be no main subprogram for the partition.
- A group communication protocol must be used for communicating with the partition.

The last restriction is an implicit one: the configuration tool gnatdist currently has no way to know whether a certain protocol is a group communication protocol.

Further extensions for specifying parameters of replication have not yet been added to the configuration language, but should be. It would for instance be convenient to be able to specify the maximum size of the event log for each replica. Currently, this value is a hard–coded constant, and whenever the event log is full, the leader replica sends it to its followers before logging a new event, even if this new event is not an observable event. By choosing the size of the event log, one can thus influence the degree of synchronization between the replicas to a certain extent.

Another extension planned is the specification of the application–defined `Get_State` and `Put_State` subprograms for state transfers in the configuration language. This would make state transfers more transparent to the application: these subprograms could be developed apart, and added to the partition only at configuration time. Currently, these subprograms must be an integral part of the application.

## 8.12 Current State

The implementation described in this chapter is not completed, alas. While the group communication protocol of PHOENIX is integrated via the daemon process described in section 8.3 and the basic support for event logging and replay of internal and external events is functional, crucial parts are still under development. In particular, state transfers do not work yet, interactions with the environment (e.g. files) are not yet handled, and the configuration language extensions are not yet complete (as mentioned above).

Event logging and replay is implemented only for events that can be intercepted within the run–time system. Some internal events cannot be handled in this way, though, and the cooperation of the compiler is required. These events are:

- Events due to assignments of controlled types. Such assignments are abort–deferred regions, but they cannot be intercepted in all cases within the run–time support. The compiler would have to generate calls to RAPIDS to log and replay such events.
- Accesses to global objects protected by pragma `Atomic`. Again, the compiler would have to translate accesses to atomic objects in such a way that the replication manager can log and replay them.
- The handling of remote accesses. The representation of remote access types should be a controlled type, and distributed objects should have a hidden component, also of a controlled type, both in order to be able to correctly garbage collect obsolete marshaled representations in the translation tables (cf. section 8.8). The compiler would have to be changed to cope with these changes of data structures.

These compiler changes have not been made: despite GNAT's availability in source form, they constitute significant changes well beyond the scope (and the time limit!) of this thesis.

The standard library has not yet been rewritten to do event logging and replay where needed.

# Chapter 9:
# Conclusion

In this final chapter, I summarize the main results and show some perspectives for future work.

## 9.1 Summary of Results

In this thesis, I have studied the problem of replica consistency in the transparent replication of non–deterministic objects (partitions) in distributed Ada 95. Partitions in Ada 95 are generally multithreaded, which causes them to exhibit a non–deterministic behavior. This non–determinism is due to deliberate underspecification in the language standard and to inherent timing dependencies in concurrent execution.

The general approach to replication I took is based on the notion of group communication: all the replicas of a partition form a group, which abstracts from and encapsulates the individual replicas. The replicated partition should behave the same way as if it was not replicated. Replication transparency is thus to be achieved not only towards the application, but preferably also (on the system level) towards other partitions. The choice of the *computation model* of the replicas greatly influences the way replicas can be organized within the group and the level of transparency that can be achieved. Two different computation models and their consequences for replication have been investigated.

The non–deterministic computation model precludes the use of an active replication scheme and necessitates the isolation of concurrent remote requests in order to avoid a domino effect. Remote procedure calls between objects must thus be implemented as *nested transactions*, similar to the interactions between guardians in Argus. It turns out that this approach is inappropriate for Ada 95: serialization of transactions may conflict with the standard semantics of the programming language, which ensures linearizability. With transactional semantics, concurrent execution of remote calls that perform partial operations may result in *unbreakable deadlocks*. As a result, the semantics of the programming language Ada 95 cannot be preserved: a partition that works correctly when not replicated might deadlock when it *is* replicated. Transactions are therefore not a suitable concept for *transparently* replicating partitions in Ada 95: a distributed application would have to be developed with this restriction in mind. Replication transparency is also compromised on the system level inasmuch as the effects of a failure are not local to the group of replicas as other partitions have to roll back remote calls originating on the partition where a replica has failed.

The *piecewise deterministic computation model* assumes that an execution consists of a sequence of deterministic state intervals and non–deterministically occurring events, where each non–deterministic event terminates a deterministic state interval and starts a new one. This model corresponds better to the semantics of the programming language and leads directly to semi–active (or leader–follower) replication of partitions. By recording the execution history, i.e. the sequence of non–deterministic events that occur, on the leader and replaying it on the followers the consistency of all replicas is guaranteed. Replica synchronization is based on the notion of *observable events*: only the leader ever interacts with other partitions of the distributed application, and the recorded history must be sent to the followers only at each such interaction. As long as the effects of the leader's execution are local to the leader itself, no synchronization is necessary. Because synchronization occurs before each interaction of the leader with the rest of the system, effects of failures remain purely local to the group of replicas: replication transparence on the system level is maintained except for the need for receiver–based duplicate message detection.

Replica transparence towards the application is only given if replicas can only fail (k–resilient objects). Recovery of replicas necessitates a state transfer from the group to the newly joining replica. Throughout this thesis, I have assumed a very general system model allowing heterogeneous distributed systems. A direct consequence of this is that the state transfer cannot be accomplished transparently in an automatic way. Instead, the application must collect and install the state itself, using routines that are invoked through callbacks from the (otherwise transparent) replication manager. Also, the concurrent nature of Ada 95 partitions restricts state transfers in heterogeneous systems to partitions that adhere to a rather restricted execution model. State transfers are only possible for partitions fulfilling the *quiescence assumption*, ruling out in particular all partitions that contain autonomous tasks.

As a feasibility study, a prototype replication manager called RAPIDS for the GNAT development system for Ada 95 has been developed. Although not completed, this implementation shows that semi–active replication can be implemented transparently in the run–time support of Ada 95 (again with the exception of state transfers).

## 9.2 Future Work

One direction of future work is certainly concerned with implementation: the current prototype should be completed, the standard library should be adapted, and the configuration tool should be extended to allow for a greater flexibility in configuring distributed applications with replicated partitions.

Another step is the implementation of the necessary compiler changes to be able to log and replay the remaining internal events: the handling of remote access types, and accesses to atomic variables.

A performance evaluation of RAPIDS still has to be done. The end–to–end performance of RAPIDS depends heavily on the performance characteristics of the group communication protocol used and hence is not of prime interest. Besides this network overhead, the performance of a replicated partition depends primarily on the speed of the event logging on the leader. An isolated performance analysis of the event logging (and replay) part might prove interesting. At the current stage, such an evaluation would be premature as RAPIDS has not at all been conceived for optimal speed — there are too many obvious places that should be optimized somewhat before engaging in serious performance analysis.

The current implementation of semi–active replication in RAPIDS contains no provisions to bound the de–synchronization of followers with respect to their leader. If a follower is significantly slower than its leader, it will receive a new event log before having replayed all of the previous extended state interval, and it will therefore lag behind by more and more extended state intervals over time. An intra–group protocol could be added to RAPIDS to bound this effect.

I also think that state transfer should be re–examined under the assumption that replicas execute (even in heterogeneous distributed systems) on a homogeneous subset of physical nodes. This assumption might alleviate some of the problems of state transfer.

Another future development might be to investigate the use of group communication at the application level, e.g. for offering “cooperative groups” in the sense of Drago.

# Part III

# Annexes

# Annex A: Bibliography


[ABHN91] Ahamad, M.; Burns, J. E.; Hutto, P. W.; Neiger, G.: "Causal Memory", in *Proceedings of the 5$^{th}$ International Workshop on Distributed Algorithms (WDAG '91)*, pp. 9 – 30, Lecture Notes in Computer Science **579**, Springer Verlag, Oct. 1991.

[AC78] Avizienis, A.; Chen, L.: "N-Version Programming: A Fault–Tolerant Approach to Reliability in Software Operation", in *Proceedings of the 8$^{th}$ International Symposium on Fault–Tolerant Computing Systems (FTCS–8)*, 1978.

[Asp98] Asplund, L. (Ed.): *Ada–Europe '98*, Uppsala, Sweden, June 1998, Lecture Notes in Computer Science **1411**, Springer Verlag, 1998.

[Avi85] Avizienis, A.: "The N–Version Approach to Fault–Tolerant Software", *IEEE Transactions on Software Engineering* ***SE–11****(6)*, Dec. 1985, pp. 1491 – 1501.

[AW94] Attiya, H.; Welch, J. L.: "Sequential Consistency versus Linearizability", *ACM Transactions on Computer Systems* ***12****(2)*, May 1994, pp. 91 – 122.

[Bar95] Barnes, J. (Ed.): *Ada 95 Rationale*, Lecture Notes in Computer Science **1247**, Springer Verlag, 1997; Intermetrics, Inc., 1995.

[BC85] Birrell, A. D.; Cheriton, D. (Eds.): *10$^{th}$ ACM Symposium on Operating System Principles*, Orcas Island WA, USA, Dec. 1985, ACM SIGOPS Operating Systems Review **19**(5).

[BDR98] Burns, A.; Dobbing, B.; Romanski, G.: "The Ravenscar Tasking Profile for High Integrity Real–Time Programs", in Asplund [Asp98], pp. 263 – 275.

[BH73] Brinch Hansen, P.: *Operating System Principles*, Prentice Hall, 1973.

[BHG87] Bernstein, P. A.; Hadzilacos, V.; Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison–Wesley, 1987.

[BHV$^{+}$90] Barrett, P. A.; Hilborne, A. M.; Veríssimo, P. et al.: "The Delta–4 Extra Performance Architecture XPA", in *Proceedings of the 20$^{th}$ International Sym-*

*posium on Fault–Tolerant Computing Systems (FTCS–20)*, pp. 481 – 488, Newcastle upon Tyne, UK, June 1990.

[Bir85] Birman, K. P.: "Replication and Fault–Tolerance in the Isis System", in Birrell; Cheriton [BC85], pp. 79 – 86.

[BJRA85] Birman, K. P.; Joseph, T. A.; Räuchle, T.; el Abbadi, A.: "Implementing Fault–Tolerant Distributed Objects", *IEEE Transactions on Software Engineering* ***SE–11****(6)*, June 1985, pp. 502 – 508.

[BL93] Bernstein, A. J.; Lewis, P. M.: *Concurrency in Programming and Database Systems*, Jones & Bartlett, 1993.

[BLS98] Black, D.; Low, C.; Shrivastava, S. K.: "The Voltan application programming environment for fail–silent processes", *Distributed Systems Engineering* ***5****(2)*, June 1998, pp. 66 – 77.

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

[BMS94] Birman, K. P.; Mattern, F.; Schiper, A. (Eds.): *Theory and Practice of Distributed Systems – International Workshop*, Dagstuhl Castle, Germany, Sept. 1994, Lecture Notes in Computer Science **938**, Springer Verlag, 1995.

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

[BR94] Birman, K. P.; van Renesse, R.: *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, 1994.

[BS93] Barrett, P. A.; Speirs, N. A.: "Towards an integrated approach to fault tolerance in Delta–4", *Distributed Systems Engineering* ***1****(2)*, 1993, pp. 59 – 66.

[BT93] Babaoglu, Ö.; Toueg, S.: "Non–Blocking Atomic Commitment", in Mullender, S. (Ed.), *Distributed Systems*, chapter 6, pp. 147 – 168, Addison–Wesley, 2$^{nd}$ ed., 1993.

[BW95] Burns, A.; Wellings, A. J.: *Concurrency in Ada*, Cambridge University Press, 1995.

[CGR88] Cmelik, R. F.; Gehani, N. H.; Roome, W. D.: "Fault–Tolerant Concurrent C: A Tool for Writing Fault–Tolerant Distributed Programs", in *Proceedings of the 18$^{th}$ International Symposium on Fault–Tolerant Computing Systems (FTCS–18)*, pp. 56 – 61, Tokyo, Japan, June 1988.

[CHT94] Chandra, T. D.; Hadzilacos, V.; Toueg, S.: "The Weakest Failure Detector for Solving Consensus". *Technical Report TR94–1426*, Department of Computer Science, Cornell University, Ithaca NY, USA, 1994.

[Coo85] Cooper, E. C.: "Replicated Distributed Programs", in Birrell; Cheriton [BC85], pp. 63 – 78.

[CPR+92] Chérèque, M.; Powell, D.; Reynier, P. et al.: "Active Replication in Delta–4", in Lala; Koren [LK92], pp. 28 – 37.

[CR86] Campbell, R. H.; Randell, B.: "Error Recovery in Asynchronous Systems", *IEEE Transactions on Software Engineering* ***SE-12****(8)*, 1986, pp. 811 – 826.

[Cri91] Cristian, F.: "Understanding Fault–Tolerant Distributed Systems", *Communications of the ACM* ***34****(2)*, Feb. 1991, pp. 56 – 78.

[CT91] Chandra, T. D.; Toueg, S.: "Unreliable failure detectors for asynchronous systems", in *Proceedings of the 10$^{th}$ Annual ACM Symposium on Principles of Distributed Computing*, pp. 325 – 340, Aug. 1991.

[CT95] Chandra, T. D.; Toueg, S.: "Unreliable Failure Detectors for Reliable Distributed Systems". *Technical Report TR95–1535*, Department of Computer Science, Cornell University, Ithaca NY, USA, 1995.

[DRAR91] Dasgupta, P.; Richard J. LeBlanc, J.; Ahamad, M.; Ramachandran, U.: "The Clouds Distributed Operating System", *Computer* ***24****(11)*, Nov. 1991, pp. 34 – 44.

[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.

[EMS91] Eppinger, J. L.; Mummert, L. B.; Spector, A. Z. (Eds.): *Camelot and Avalon — A Distributed Transaction Facility*, Morgan Kaufmann, 1991.

[EZ92a] Elnozahy, E. N.; Zwaenepoel, W.: "Implementation and Performance of Transparent Rollback–Recovery in Manetho". *Technical Report TR92–197*, Rice University, Houston TX, USA, 1992.

[EZ92b] Elnozahy, E. N.; Zwaenepoel, W.: "Replicated Distributed Processes in Manetho", in Lala; Koren [LK92], pp. 18 – 27.

[EZ94] Elnozahy, E. N.; Zwaenepoel, W.: "On the Use and Implementation of Message Logging", in *Proceedings of the 24$^{th}$ International Symposium on Fault–Tolerant Computing Systems (FTCS–24)*, pp. 298 – 307, June 1994.

[FHZ97] de Ferreira Rezende, F.; Härder, T.; Zielinski, J.: "Transaction Identifiers in Nested Transactions: Implementation Schemes and Performance", *The Computer Journal* ***40****(5)*, 1997, pp. 245 – 258.

[FLP85] Fischer, M.; Lynch, N. A.; Paterson, M.: "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM* ***32****(2)*, 1985, pp. 374 – 382.

[FLW92] Fekete, A.; Lynch, N. A.; Weihl, W. E.: "Hybrid Atomicity for Nested Transactions", in *Proceedings of the 4th International Conference on Database Theory (ICDT '92)*, pp. 216 – 230, Berlin, Germany, Oct. 1992, Lecture Notes in Computer Science **646**, Springer Verlag, 1991.

[GBCR93] Glade, B. B.; Birman, K. P.; Cooper, R. C.; van Renesse, R.: "Light–weight process groups in the Isis system", *Distributed Systems Engineering* ***1****(1)*, 1993, pp. 29 – 36.

[Ghe90] Ghemawat, S.: *Automatic Replication for Highly Available Services*, MS Thesis MIT-LCS-TR-473, Laboratory for Computer Science, MIT, Cambridge MA, USA, Jan. 1990.

[GLS95] Guerraoui, R.; Larrea, M.; Schiper, A.: "Non–Blocking Atomic Commitment with an Unreliable Failure Detector", in *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pp. 41 – 50, Bad Neuenahr, Germany, Sept. 1995, IEEE Computer Society Press, 1995.

[GMAA97] Guerra, F.; Miranda, J.; Alvarez, A.; Arévalo, S.: "An Ada Library to Program Fault–Tolerant Distributed Applications", in *Proceedings of Ada–Europe '97*, pp. 230 – 243, London, UK, June 1997, Lecture Notes in Computer Science **1251**, Springer Verlag, 1997.

[GR96] Garg, V. K.; Raynal, M.: "Normality: A Consistency Condition for Concurrent Objects". *Technical Report PI–1015*, IRISA, Rennes, France, May 1996.

[Gra78] Gray, J. N.: "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course*, pp. 393 – 481, Berlin, 1978, Lecture Notes in Computer Science **60**, Springer Verlag, 1978.

[GS94] Guerraoui, R.; Schiper, A.: "Transaction model vs Virtual Synchrony model: bridging the gap", in Birman et al. [BMS94], pp. 121 – 132.

[GS96] Guerraoui, R.; Schiper, A.: "Fault Tolerance by Replication in Distributed Systems", in Strohmeier [Str96], pp. 38 – 57.

[HGC97] de las Heras-Quirós, P.; González-Barahona, J. M.; Centeno-González, J.: "Programming Distributed Fault–Tolerant Systems: The replicAda Approach", in *Proceedings of Tri-Ada '97*, pp. 21 – 29, St. Louis MO, USA, Nov. 1997.

[HLMR74] Horning, J. J.; Lauer, H. C.; Melliar-Smith, P.; Randell, B.: "A Program Structure for Error Detection and Recovery", in *Lecture Notes in Computer Science* ***16***, pp. 177 – 193, Springer Verlag, Springer Verlag, 1974.

[Hoa74] Hoare, C. A. R.: "Monitors: An Operating System Structuring Concept", *Communications of the ACM* ***17****(10)*, Oct. 1974, pp. 549 – 557.

[HR83] Härder, T.; Reuter, A.: "Principles of Transaction–Oriented Database Recovery", *ACM Computing Surveys* ***15****(4)*, 1983, pp. 287 – 317.

[HSL78] Hopkins, A. L., Jr.; Smith, T. B., III; Lala, J. H.: "FTMP: A Highly Reliable Fault–Tolerant Multiprocessor for Aircraft", *Proceedings of the IEEE* ***66****(10)*, Oct. 1978, pp. 1221 – 1239.

[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.

[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.

[JC86] Jalote, P.; Campbell, R. H.: "Atomic Actions for Software Fault Tolerance using CSP", *IEEE Transactions on Software Engineering* ***SE–12****(1)*, 1986, pp. 59 – 68.

[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.

[Kim95] Kim, K. H.: "The Distributed Recovery Block Scheme", in Lyu [Lyu95], chapter 8, pp. 189 – 209.

[KNP96] Kermarrec, Y.; Nana, L.; Pautet, L.: "GNATDIST: a configuration language for distributed Ada 95 applications", in *Proceedings of Tri-Ada '96*, pp. 63 – 72, Philadelphia PA, USA, Dec. 1996.

[Kop97] Kopetz, H.: *Real–Time Systems — Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.

[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.

[LABK90] Laprie, J.-C.; Arlat, J.; Béounes, C.; Kanoun, K.: "Definition and Analysis of Hardware– and Software–Fault–Tolerant Architectures", *IEEE Computer **23**(7)*, July 1990, pp. 39 – 51.

[Lam78] Lamport, L.: "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM **21**(7)*, July 1978, pp. 558 – 565.

[Lam79] Lamport, L.: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers **C–28**(9)*, Sept. 1979, pp. 690 – 691.

[Lap85] Laprie, J.-C.: "Dependable Computing and Fault Tolerance : Concepts and Terminology", in *Proceedings of the 15th International Symposium on Fault–Tolerant Computing Systems (FTCS–15)*, pp. 2 – 11, Ann Arbour MI, USA, June 1985.

[LCJS87] Liskov, B. H.; Curtis, D.; Johnson, P.; Scheifler, R.: "Implementation of Argus", in *Proceedings of the 11th ACM Symposium on Operating System Principles*, pp. 111 – 122, Austin TX, USA, Nov. 1987, ACM SIGOPS Operating Systems Review **21**(5).

[Led95] Ledru, P.: *Distributed Programming in Ada with Protected Objects*, MS Thesis, Dept. of Computer Science, University of Alabama, Huntsville AL, USA, Nov. 1995.

[Lis85] Liskov, B. H.: "The Argus Language and System", in *Distributed Systems: Methods and Tools for Specification — An Advanced Course*, pp. 342 – 430, Lecture Notes in Computer Science **190**, Springer Verlag, 1985.

[Lis88] Liskov, B. H.: "Distributed Programming in Argus", *Communications of the ACM **31**(3)*, Mar. 1988, pp. 300 – 312.

[LJP93] Lea, R.; Jacquemot, C.; Pillevesse, E.: "COOL: System Support for Distributed Programming", *Communications of the ACM* ***36****(9)*, Sept. 1993, pp. 37 – 46.

[LK92] Lala, J. H.; Koren, I. (Eds.): *22$^{nd}$ International Symposium on Fault–Tolerant Computing Systems (FTCS–22)*, Boston MA, USA, July 1992.

[LS83] Liskov, B. H.; Scheifler, R.: "Guardians and Actions: Linguistic Support for Robust Distributed Programs", *ACM Transactions on Programming Languages and Systems* ***5****(3)*, 1983, pp. 381 – 404.

[LS90] Little, M. C.; Shrivastava, S. K.: "Replicated K–Resilient Objects in Arjuna", in *Proceedings of the 1$^{st}$ IEEE Workshop on Replicated Data*, Houston TX, USA, Nov. 1990.

[LSP82] Lamport, L.; Shostak, R.; Pease, M.: "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems* ***4****(3)*, 1982, pp. 382 – 401.

[LTBL97] Litzkow, M.; Tannenbaum, T.; Basney, J.; Livny, M.: "Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System". *Technical Report TR–97–1346*, Computer Sciences Department, University of Wisconsin, Madison WI, USA, 1997.

[Lyn83] Lynch, N. A.: "Concurrency Control for Resilient Nested Transactions", in *Proceedings of the 2$^{nd}$ ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pp. 166 – 181, Atlanta GA, USA, Mar. 1983.

[Lyu95] Lyu, M. R. (Ed.): *Software Fault Tolerance*, John Wiley & Sons, 1995.

[MAAG96] Miranda, J.; Alvarez, A.; Arévalo, S.; Guerra, F.: "Drago: An Ada Extension to Program Fault–Tolerant Distributed Applications", in Strohmeier [Str96], pp. 235 – 246.

[Mad98] Madria, S. K.: "A Study of the Concurrency Control and Recovery Algorithms in Nested Transaction Environment", *The Computer Journal* ***40****(10)*, 1998, pp. 630 – 639.

[Mal96] Malloth, C.: *Conception and Implementation of A Toolkit for Building Fault–Tolerant Distributed Applications in Large–Scale Networks*, PhD Thesis #1557, Dept. of Computer Science, Swiss Federal Institute of Technology, Sept. 1996.

[Maz96] Mazouni, K.: *Étude de l'invocation entre objets dupliqués dans un système réparti tolérant aux fautes*, PhD Thesis #1578, Dept. of Computer Science, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1996.

[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.

[MHL+92] Mohan, C.; Haderle, D.; Lindsay, B.; Pirahesh, H.; Schwarz, P.: "ARIES: A Transaction Recovery Method Supporting Fine–Granularity Locking and Partial Rollbacks Using Write–Ahead Logging", *ACM Transactions on Database Systems **17**(1)*, Mar. 1992, pp. 94 – 162.

[ML83] Mohan, C.; Lindsay, B.: "Efficient Commit Protocols for the "Tree of Processes" Model of Distributed Transactions", in *Proceedings of the 2$^{nd}$ ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug. 1983.

[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.

[Mos85] Moss, J. E. B.: *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[Mos93] Mosberger, D.: "Memory Consistency Models". *Technical Report TR 93/11*, University of Arizona, Tucson AZ, USA, 1993.

[MW91] Mössenböck, H.; Wirth, N.: "The Programming Language Oberon–2". *Technical Report 160*, Institut für Computersysteme, ETH Zürich, Switzerland, May 1991.

[MWR98] Mitchell, S.; Wellings, A. J.; Romanovsky, A.: "Distributed Atomic Actions in Ada 95". *Technical Report YCS–98–298*, Dept. of Computer Science, University of York, UK, 1998.

[Oki88] Oki, B. M.: *Viewstamped Replication for Highly Available Distributed Systems*, PhD Thesis MIT-LCS-TR-423, Laboratory for Computer Science, MIT, Cambridge MA, USA, May 1988.

[Pac95] Pacull, F.: *Concepts et méchanismes pour la mise en oeuvre d'un environnement d'édition coopérative sur un réseau à grande échelle*, PhD Thesis #1335, Dept. of Computer Science, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1995.

[Pap79] Papadimitriou, C. H.: "The Serializability of Concurrent Database Updates", *Journal of the ACM **26**(4)*, Oct. 1979, pp. 631 – 653.

[PBKL94] Plank, J. S.; Beck, M.; Kingsley, G.; Li, K.: "libckpt: Transparent Checkpointing under Unix". *Technical Report CS–TR–94–242*, Department of Computer Science, University of Tennessee, Knoxville TN, USA, Aug. 1994.

[PCD90] Powell, D.; Chérèque, M.; Drackley, D.: "Fault–Tolerance in Delta–4", in *Proceedings of the 4th ACM SIGOPS Workshop on Fault Tolerance Support in Distributed Systems*, pp. 122 – 125, Bologna, Italy, Sept. 1990, ACM SIGOPS Operating Systems Review **25**(2).

[PJA98] Patiño-Martinez, M.; Jiménez-Peris, R.; Arévalo, S.: "Integrating Groups and Transactions: A Fault–Tolerant Extension of Ada", in Asplund [Asp98], pp. 78 – 89.

[PL96] Pruyne, J.; Livny, M.: "Managing Checkpoints for Parallel Programs", in *Proceedings of the IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 140 – 154, Honolulu HI, USA, Apr. 1996.

[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.

[Pow94] Powell, D.: "Distributed Fault Tolerance: Lessons from Delta–4", *IEEE Micro **14**(1)*, Feb. 1994, pp. 36 – 47.

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

[PS88] Parrington, G. D.; Shrivastava, S. K.: "Implementing Concurrency Control in Reliable Distributed Object–Oriented Systems", in *Proceedings of the 2nd European Conference on Object–Oriented Programming (ECOOP '88)*, pp. 243 – 249, Oslo, Norway, Aug. 1988.

[PSWL95] Parrington, G. D.; Shrivastava, S. K.; Wheater, S. M.; Little, M. C.: "The Design and Implementation of Arjuna", *Usenix Computing Systems Journal **8**(3)*, 1995.

[PT98] Pautet, L.; Tardieu, S.: "Inside the Distributed Systems Annex", in Asplund [Asp98], pp. 65 – 77.

[RA94] Resende, R. F.; Abbadi, A. E.: "On the serializability theorem for nested transactions", *Information Processing Letters* ***50****(4)*, May 1994, pp. 177 – 183.

[Ran75] Randell, B.: "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering* ***SE–1****(2)*, 1975, pp. 220 – 232.

[RBM96] van Renesse, R.; Birman, K. P.; Maffeis, S.: "Horus: A Flexible Group Communication System", *Communications of the ACM* ***39****(4)*, Apr. 1996.

[Ree78] Reed, D. P.: *Naming and Synchronization in a Decentralized Computer System*, PhD Thesis MIT/LCS/TR–205, MIT, Cambridge MA, USA, Sept. 1978.

[RS96] Ramkumar, B.; Strumpen, V.: "Portable Checkpointing and Recovery in Heterogeneous Environments". *Technical Report ECE–96-6-1*, Department of Electrical and Computer Engineering, University of Iowa, Iowa City IA, USA, 1996.

[RS97] Ramkumar, B.; Strumpen, V.: "Portable Checkpointing for Heterogeneous Architectures", in *Proceedings of the 27$^{th}$ International Symposium on Fault–Tolerant Computing Systems (FTCS–27)*, pp. 58 – 67, Seattle WA, USA, June 1997.

[RX95] Randell, B.; Xu, J.: "The Evolution of the Recovery Block Concept", in Lyu [Lyu95], chapter 1, pp. 1 – 21.

[SB89] Speirs, N. A.; Barrett, P. A.: "Using Passive Replication in Delta–4 to Provide Dependable Distributed Computing", in *Proceedings of the 19$^{th}$ International Symposium on Fault–Tolerant Computing Systems (FTCS–19)*, pp. 184 – 190, Chicago IL, USA, June 1989.

[SB94] Schonberg, E.; Banner, B.: "The GNAT Project: A GNU–Ada 9X Compiler", in *Proceedings of Tri-Ada '94*, pp. 48 – 57, Baltimore MD, USA, Nov. 1994.

[SC91] Seaton, D.; Chérèque, M.: "Input/Output: Interfacing the Real World", in Powell [Pow91], chapter 12, pp. 307 – 328.

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

[SD$^+$85] Spector, A. Z.; Daniels, D. et al.: "Distributed Transactions for Reliable Systems", in Birrell; Cheriton [BC85], pp. 127 – 146.

[SDP91] Shrivastava, S. K.; Dixon, G. N.; Parrington, G. D.: "An Overview of the Arjuna Distributed Programming System", *IEEE Software* ***8****(1)*, Jan. 1991.

[SE+87] Spector, A. Z.; Eppinger, J. L. et al.: "High Performance Distributed Transaction Processing in a General Purpose Computing Environment", in *Proceedings of the 2nd International Workshop on High–Performance Transaction Systems*, pp. 220 – 242, Pacific Grove CA, USA, Sept. 1987, Lecture Notes in Computer Science **359**, Springer Verlag, 1987.

[SEST92] Shrivastava, S. K.; Ezhilchelvan, P. D.; Speirs, N. A.; Tully, A.: "Principal Features of the Voltan Family of Reliable Node Architectures for Distributed Systems", *IEEE Transactions on Computers **41**(5)*, May 1992, pp. 542 – 549.

[SF98] Sens, P.; Folliot, B.: "The STAR Fault Manager for Distributed Operating Environments: Design, Implementation, and Performance", *Software — Practice & Experience **28**(10)*, Aug. 1998, pp. 1079 – 1100.

[Shr94] Shrivastava, S. K.: "Lessons Learned from Building and Using the Arjuna Distributed Programming System", in Birman et al. [BMS94].

[SMR93] Shrivastava, S. K.; Mancini, L. V.; Randell, B.: "The Duality of Fault–Tolerant System Structures", *Software — Practice & Experience **23**(7)*, July 1993, pp. 773 – 798.

[SR96] Schiper, A.; Reynal, M.: "From Group Communications to Transactions in Distributed Systems", *Communications of the ACM **39**(4)*, Apr. 1996.

[Str96] Strohmeier, A. (Ed.): *Ada–Europe '96*, Montreux, Switzerland, June 1996, Lecture Notes in Computer Science **1088**, Springer Verlag, 1996.

[Str98] Strumpen, V.: "Compiler Technology for Portable Checkpoints", submitted for publication, *http://theory.lcs.mit.edu/~strumpen/porch.ps.gz*, 1998.

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

[Tan95] Tanenbaum, A. S.: *Distributed Operating Systems*, Prentice Hall, 1995.

[VBB+91] Veríssimo, P.; Barrett, P. A.; Bond, P.; Hilborne, A. M.; Rodrigues, L.; Seaton, D.: "Extra Performance Architecture (XPA)", in Powell [Pow91], chapter 9, pp. 211 – 266.

[WB96] Wellings, A. J.; Burns, A.: "Programming Replicated Systems in Ada 95", *The Computer Journal **39**(5)*, 1996, pp. 361 – 373.

[Wei88] Weikum, G.: *Transaktionen in Datenbanksystemen*, Addison–Wesley, 1988.

[Wei89] Weihl, W. E.: “Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types”, *ACM Transactions on Programming Languages and Systems* ***11**(2)*, Apr. 1989, pp. 249 – 283.

[Wei93] Weihl, W. E.: “The Impact of Recovery on Concurrency Control”, *Journal of Computer and System Sciences* ***47***, 1993, pp. 157 – 184.

[Wir88a] Wirth, N.: “Type Extensions”, *ACM Transactions on Programming Languages and Systems* ***10**(2)*, Apr. 1988, pp. 204 – 214.

[Wir88b] Wirth, N.: “The Programming Language Oberon”, *Software — Practice & Experience* ***18**(7)*, July 1988, pp. 671 – 690.

[WL85] Weihl, W. E.; Liskov, B. H.: “Implementation of Resilient, Atomic Data Types”, *ACM Transactions on Programming Languages and Systems* ***7**(2)*, Apr. 1985, pp. 244 – 269.

[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 $8^{th}$ International Real–Time Ada Workshop*, pp. 106 – 110, Ravenscar, UK, Apr. 1997, ACM Ada Letters **XVII**(5).

# Annex B: Author and Citation Index

## L



## M

## R



## S

# Curriculum Vitae

I was born in 1965 in Schaffhausen, Switzerland, and grew up in the nearby village of Thayngen. In 1985 I completed college with a "Matura Typus C" (scientific bias) and went on to Zurich, where I studied computer science at the Swiss Federal Institute of Technology (Eidgenössische Technische Hochschule Zürich, ETHZ). After five years of intensive studies I received the MS degree/engineering diploma (Dipl. Informatik–Ingenieur ETH) in 1991. My diploma thesis, which was supervised by Prof. Niklaus Wirth, was entitled "M2: Ein Modula–2 Front–End für OP2" and dealt with the development of a Modula–2 compiler for the Oberon system.

From 1991 to 1995 I worked in a small company called HIWARE AG as a senior software engineer and developer, developing ANSI–C and Modula–2 cross–compilers for embedded systems. After four years, I decided to quit and to go back to university to pursue a PhD.

Since 1995, I have been working as a research and teaching assistant at the Laboratoire de Génie Logiciel (Software Engineering Lab) at the Swiss Federal Institute of Technology in Lausanne (Ecole Polytéchnique Fédérale de Lausanne, EPFL). Since autumn 1998, I am a lecturer at EPFL.