The GARF Library of DSM Consistency Models *

Extended Abstract

Rachid Guerraoui, Benoît Garbinato, Karim R. Mazouni
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015, Suisse
e-mail: guerraoui@di.epfl.ch

GARF is an object-oriented system intended to support high level and modular programming of reliable distributed applications. GARF provides a *distributed shared memory* abstraction, but does not enforce any particular *consistency criterion*. GARF rather offers an extensible library of consistency criteria. The programmer can bind a specific criterion to each object, according to its use and semantics. This flexible approach enables to increase availability of objects of which behaviors do not require strong consistency guarantees.

1 Introduction

Shared memory is an attractive programming paradigm to represent communication among distributed computing entities. It has been widely used for many years, and is at a higher abstraction level than the *message passing* paradigm. Providing the shared memory abstraction on top of a distributed architecture has lead to the so called *Distributed Shared Memory* (DSM) paradigm. An important issue in implementing DSM is to preserve consistency of shared and possibly replicated objects despite concurrency and failures. To address this issue, many *consistency criteria* have been proposed in the literature.

Among criteria, some guarantee *strong consistency* by ensuring a behavior similar to a centralized system. *Linearizability* [6] and *sequential consistency* [8] (or *serializability*) are the two famous examples of *strong consistency* criteria. It has been shown that these criteria cannot be implemented efficiently and do not tolerate failures.

- Lipton and Sandberg showed in [10] that, if *strong consistency* is to be guaranteed, the response time of shared object invocations will depend on the number of nodes in the system and the communication delay between these nodes.
- Herlihy showed in [6] that, if *strong consistency* is to be guaranteed, objects of complex types, such as *queue, list*, and *stack*, will not tolerate failures in an asynchronous system.

Other criteria guarantee only *weak consistency*, and do not ensure a similar behavior to a centralized system. Some examples of weak criteria are *slow consistency* [7], *PRAM consistency* [10], and *causal consistency* [7].

*Appears in proc ACM SIGOPS European workshop on Operating Systems (pages 51-57), 1995. Research funded by the Fonds National Suisse de la Recherche Scientifique as part of the Programme Prioritaire Informatique (SPP IF) under project number 5003-034344.

All these criteria are scalable, can be implemented efficiently, and tolerate node failures as well as link failures. Nevertheless, they can only be used for objects with weak consistency requirements. We advocate a flexible scheme where the consistency criterion can be customized for each object according to its use and semantics [3], i.e. there is no unique criterion for all objects. We believe that for some objects, strong consistency is necessary, whereas for others, performance or fault-tolerance are more important than strong consistency.

This paper describes the modular approach we have adopted in the GARF [4,5] object-oriented distributed system to enable the mapping of objects with different consistency requirements to different criteria. GARF provides an extensible library of consistency criteria. The programmer can choose from this library, the adequate criterion to be applied for each object. An experienced programmer can also extend the library by adding new criteria. The reminder of the paper is organised as follows. Next section presents a brief overview of GARF. Section 3 describes the GARF library of consistency criteria. Section 4 concludes with some general remarks about related work, implementation, and some experiment.

2 Overview of GARF

GARF is an object-oriented programming environment aimed to support a modular design and implementation of reliable distributed applications [4,5]. GARF supplies three programming levels, called respectively, the *functional* level, the *behavioral* level, and the *system* level.

1. At the *functional* level, application functionalities are described using passive objects, named *data objects*, in a centralized and sequential environment.
2. At the *behavioral* level, data objects are dynamically bound to *encapsulators* and *mailers*. GARF offers a library of encapsulator classes, and a library of mailer classes. Each of these classes (called *behavioral classes*) represents a specific *distribution-related* abstraction.
3. At the *system level*, programmers can extend the library of behavioral classes using low level mechanisms for remote invocation, group management, etc. These mechanisms are represented in GARF by a library of *system* classes.

At the behavioral level, each data object *o* can be bound to one encapsulator *enc(o)*. Both *o* and *enc(o)* are always located on the same node. Every invocation *on* and *from o* goes through *enc(o)*. Hence *enc(o)* controls how *o sends* and *receives* invocations. GARF provides encapsulator classes to handle *asynchrony, concurrency control, replication (passive or active), etc.* These classes are presented in figure 1 (a) and detailed in [4]. For example, the encapsulator class *ActReplica* provides support for active replication. When a data object *o* is bound to an encapsulator *enc(o)* from the class *ActReplica*, both *o* and *enc(o)* are automatically replicated on a set of nodes given as a parameter by the programmer. Every invocation on *o* goes through all replicas of *enc(o)*, and is treated by all replicas of *o*.

Each data object *o* bound to an encapsulator is also bound to a mailer class *Ma(o)*. A mailer *ma(o)* of this class is created on each node *from* which *o* is invoked; *ma(o)* is in charge of (remotely) transmitting the invocation to *enc(o)* and returning back the reply. If *o* and *enc(o)* are replicated, *ma(o)* is in charge of multicasting invocations to replicas of *enc(o)* according to some ordering semantics specified in *Ma(o)*. Mailers can be viewed as *intelligent* proxies [11], hiding distribution and replication from the clients (i.e., providing the DSM abstraction). Among mailer classes provided by GARF (figure 1 (b)), some represent consistency criteria (these are subclasses of the class *Mcast* on figure 1 (b)). The programmer can associate a specific criterion to a data object by binding it to the appropriate mailer class. This binding process (described in [5]) is done in a modular way since data objects, encapsulators, and mailers are clearly separated.

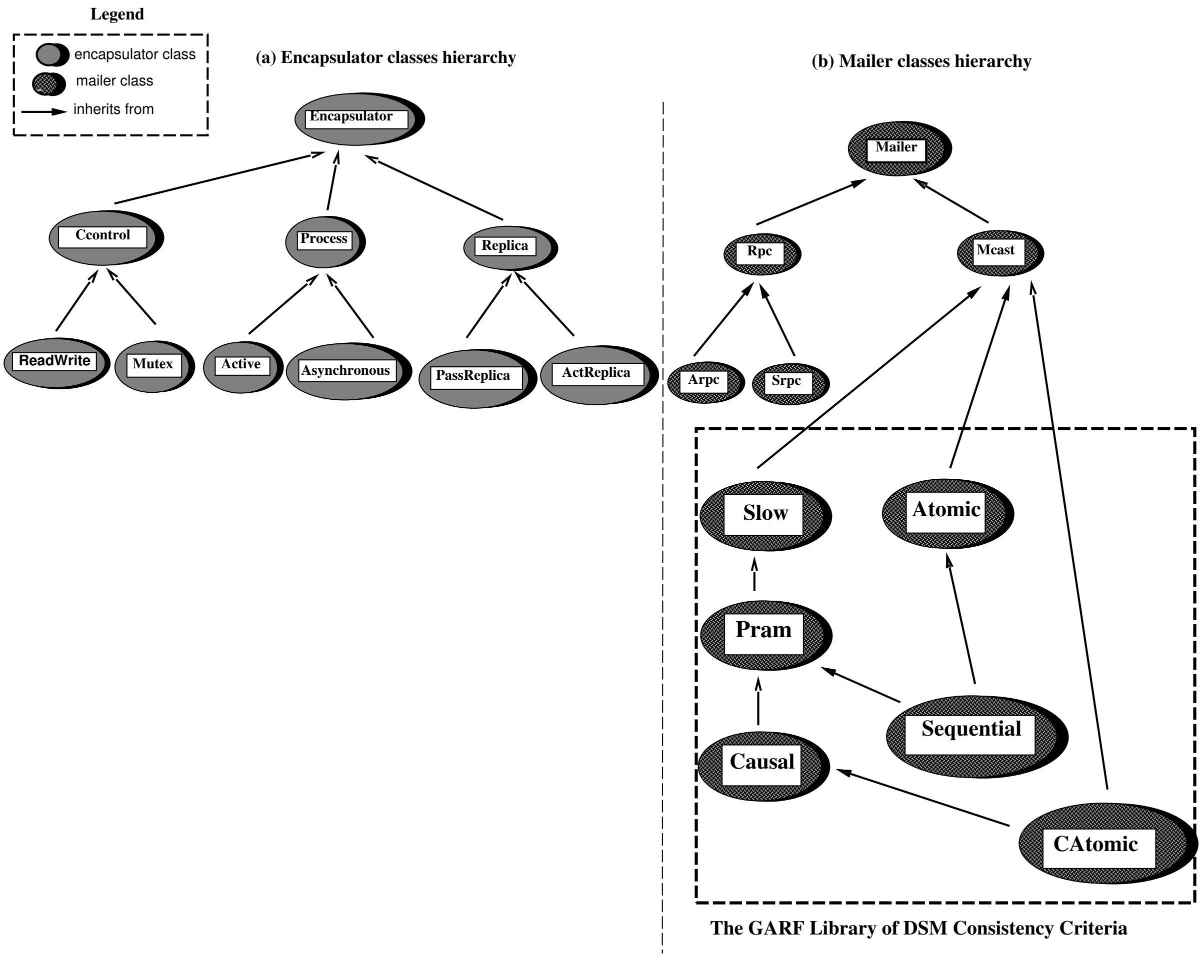

Figure 1: The GARF libraries of behavioral classes

3 The Library of Consistency Criteria

The three mailer classes *Atomic*, *Sequential*, and *CAtomic* represent strong consistency criteria. In other words, every object o which requires strong consistency must be bound to one of these mailer classes. As a consequence of this binding, a *one-copy equivalence* will be ensured for the executions of o's replicas, i.e. all o's replicas will perform invocations in the same order[1]. To achieve such a behavior, each of the classes *Atomic*, *Sequential*, and *CAtomic*, implements a totally ordered multicast primitive to transmit invocations to replicas. The main difference between these classes stands in the order preserved by the multicast primitive.

- Mailers of the class *Atomic* ensure only that all replicas treat invocations in the same order (whatever the order is). If two clients *c1* and *c2* perform two invocations *inv1* and *inv2* on a shared replicated object o, bound to the mailer class *Atomic*, all o's replicas will treat *inv1* and *inv2* in the same order.
- Mailers of the class *Sequential* guarantee the behavior provided by the class *Atomic*, but preserve in addition the *fifo* order: that is, two invocations *inv1* and *inv2* invoked by the same client in the order *inv1;inv2* will be treated in the order *inv1;inv2*. The behavior of the class *Sequential* corresponds to the *sequential consistency* criteria [8]. The class *Sequential* is a subclass of both classes *Atomic* and *Pram* (see later).
- Mailers of the class *CAtomic* guarantee the behavior provided by the class *Sequential*, but preserve in addition the *causal* order: that is, two invocations (either from the same client or not) that are causally related are treated in their causal order. The behavior of the class *CAtomic* corresponds to a specific interpretation of the *linearizability* criteria[2]. The class *CAtomic* is a subclass of both classes *Sequential* and *Causal* (see later).

The consistency criteria provided by the classes *Atomic*, *Sequential*, and *CAtomic* are only necessary for objects with strong consistency requirements, otherwise they hamper performance and fault-tolerance. For example, it is not possible for two partitions of a distributed system to access different replicas of the same strongly consistent object and reconnect later. One of the partitions must block and wait for the partition repair. This is due to the total order requirement.

The three mailer classes *Slow*, *Pram*, and *Causal* represent weak consistency criteria. An object which does not require strong consistency can be bound to one of these classes. None of these classes ensures a total ordering treatment, i.e. a one-copy equivalence.

- Mailers of the class *Slow* ensure only a *weak fifo* order: two invocations from the same client on the same object are treated in the same order. The behavior of the class *Slow* corresponds to the *slow consistency* criteria [7]. The class *Slow* is a subclass of the class *Mcast*.
- Mailers of the class *Pram* ensure a *stronger fifo* order: two invocations from the same client are treated in the same order (although the invocations concern different objects). The behavior of the class *Pram* corresponds to the *PRAM consistency* criteria [10]. The class *PRAM* is a subclass of the class *Slow*.
- Mailers of the class *Causal* ensure that two invocations that are causally ordered (either from the same client or not) are treated in their causal order. The behavior of the class *Causal* corresponds to the *causal consistency* criteria [7], which can be viewed as the transitive closure of the *fifo* property provided by *PRAM* consistency. The class *Causal* is a subclass of the class *Pram*.

[1] We consider only deterministic objects.

[2] *Linearizability* [6] requires preserving real-time, but in a (loosely-coupled) distributed system, real-time can be assimilated to causality.

Each of the classes *Slow, Pram, Causal* has an efficient and scalable implementation in the sense that the response time of an invocation depends neither on the communication delay between distributed nodes, nor on the number of these nodes. In addition, these implementations can tolerate failures. For example, replicas of the same object can treat invocations and return replies in different partitions. These replicas can then be reconnected later (if the partition is repaired) violating neither the *slow consistency*, nor the *PRAM consistency*, nor the *causal consistency* criteria.

4 Concluding remarks

Most distributed systems that supply a distributed shared memory abstraction enforce a unique consistency criterion, i.e. the criterion is the same for all objects [2,9]. In general, these systems provide a strong criterion. The Munin distributed system enables *type-specific coherence mechanism*. That is, each object, according to its use, e.g. *write-once*, *private*, etc, can provide a particular consistency mechanism. GARF follows Munin flexible approach, but differs in supporting a higher level extensible library of criteria. The GARF binding process between an object and a consistency criterion is performed in a dynamic and modular way.

We have implemented a GARF prototype on a distributed extension of Smalltalk-80 on top of the ISIS tool-kit. We have used ISIS process group management facilities to implement replicated objects, and ISIS multicast primitives to implement consistency criteria. The mailer class *Pram* uses the ISIS primitive *fbcast* which ensures that messages are received by all processes in the order they were sent. The mailer class *Causal* uses the ISIS primitive *cbcast* which ensures that messages that are causally related are received by all processes according to their causal order. Finally, the mailer class *CAtomic* uses the ISIS primitive *abcast* which ensures that messages are performed in a total order preserving causality. Other mailer classes are not yet implemented in the current GARF prototype.

We are currently experimenting with heterogeneous consistency criteria within a reliable distributed diary manager application written with GARF. In this application, strong consistency is required for *diary* objects, but only weak consistency is required for the *diary index* object (i.e. the dictionary). Indeed, operations *register*[3] and *lookup* of class *DiaryIndex* do not require total order execution on different replicas.

References

[1] J. Bennett, J. Carter, and W. Zwaenepoel - Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence - In ACM Proc of the Symp of Principles and Practices on Parallel Programming, pp. 168-176 - 1990.

[2] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield - The Amber System: Parallel programming on a network of multiprocessors - In ACM Proc of the Symp on Operating Systems Principles, pp 147-158 - 1989.

[3] D. Cheriton - Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design - IEEE Proc of the Int Conference on Distributed Computing Systems, pp. 190-197 - 1986.

[4] B. Garbinato, R. Guerraoui, and K. Mazouni - Distributed Programming in GARF - In Object Based Distributed Programming - R. Guerraoui, O. Nierstrasz, and M. Riveill eds - LNCS 791, Springer, pp 225-239 - 1994.

[3] Given the assumption that different users will never register with the same name.

[5] R. Guerraoui, B. Garbinato, and K. Mazouni - Overview of GARF Design and Implementation - Internal Report, LSE/DI/EPFL (submitted for publication) - 1994.

[6] M. Herlihy - Wait-Free Synchronization - ACM Transactions on Programming Languages And Systems, 11 (1) pp 124-149 - 1991.

[7] P. Hutto, and M. Ahamad - Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories - In IEEE Proc of the Int Conf on Distributed Computing Systems, pp. 302-309 - 1990.

[8] L. Lamport - On interprocess communication; Part I: Basic formalism - Distributed Computings 1 (7) pp 77-85 - 1986.

[9] K. Li - Shared Virtual Memory on Loosely Coupled Multiprocessors - PhD Thesis, Yale University - 1986.

[10] R. Lipton and J. Sandberg - PRAM: a scalable shared memory - Technical Report CS-TR-180-88, Princeton Univ, Dept of Computer Science - September 1988.

[11] M. Shapiro - Structure and Encapsulation in Distributed System: The Proxy Principle - IEEE Proc of the Int Conference on Distributed Computing Systems, pp. 198-205 - 1986.