Distributed Programming in GARF *

Benoît Garbinato, Rachid Guerraoui, Karim R. Mazouni

Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Suisse

Abstract. GARF is an object-oriented programming environment aimed to support the design of reliable distributed applications. Its computational model is based on two programming levels: the *functional* level and the *behavioral* level. At the *functional* level, software functionalities are described using passive objects, named *data objects*, in a centralized, volatile, and failure free environment. At the *behavioral* level, data objects are dynamically bound to *encapsulators* and *mailers* which support distribution, concurrency, persistence and fault tolerance. Encapsulators wrap data objects by controlling how the latter send and receive messages, while mailers perform communications between encapsulators. This paper describes how the GARF computational model enables to build flexible and highly modular abstractions for the design of reliable distributed applications.

1 Introduction

Transparency has been recognized to be a worthwhile goal in the design of distributed systems. A user of a transparently distributed system could make no distinction between local resource accesses and remote ones. From the point of view of a distributed application designer, we believe that transparency has to be provided "à la carte". That is, a distributed programming environment must provide programmers with a set of *behavioral* abstractions that hides low level features related to concurrency, persistence, distribution and fault-tolerance. Nevertheless, programmers must be able to refine such abstractions in order to obtain the desired efficiency. For example, the underlying system may decide to locate, by default, the application's components on different nodes, according to a particular load balancing policy. However, the actual distribution of the program is sometimes likely to be closely associated with the semantics of the program. Therefore, programmers must be able to explicitly locate components. There are many other examples related to concurrency and replication control where default behaviors are too restrictive and where it is often desirable to use application's semantics to refine these behaviors.

In this paper, we describe the computational model of GARF: an object-oriented distributed system that supports transparency "à la carte". GARF provides built-in *behavioral* abstractions (i.e., abstractions that deal with concurrency, persistence, distribution, and fault-tolerance), which can be used as such, or refined at programmers' convenience.

GARF promotes software modularity by separating the *functional* aspects of applications, from their *behavioral* features. The former are designed within *data objects*, similar to objects

* Research supported by the Swiss FNS as part of the project Mesures Spéciales under contract number 5003-034344.

in classical centralized languages, whereas the latter are confined within particular objects named *encapsulators* and *mailers*. *Encapsulators* are used to control how data objects send and receive messages, whereas *mailers* are used to perform communications between encapsulators. GARF provides a built-in extensible library of encapsulator and mailer classes that implement very flexible behavioral features such as client/server asynchrony, active/passive replication, semantic-based concurrency control, etc. These classes can be used as such, or refined according to application's semantics.

The remainder of this paper is organized as follows. Section 2 describes the GARF computational model. Section 3 shows its flexibility and modularity through some distribution related examples. Section 4 presents a simple distributed and fault-tolerant application implemented in GARF. Section 5 contrasts GARF with related work and section 6 concludes with some general remarks.

2 The GARF computational model

2.1 Two programming levels

The GARF computational model is based on two programming levels. A *high* level, called *functional level*, in which the programmer describes aspects of the application that could be expressed in a centralized, volatile, and sequential object-oriented system; and a *low* level, called *behavioral level*, in which programmer describes behavioral features related to concurrency, persistence, distribution, and fault-tolerance.

The functional level: at this level, the programmer focuses on all sequential aspects of the application, assuming a volatile, centralized, and failure-free environment. Such (functional) aspects are described through classes, called *data classes*, of which instances are called *data objects*. The programmer can here reuse classes designed within a classical object-oriented language (Smalltalk in our current prototype).

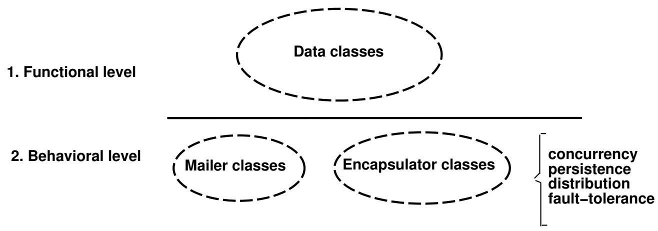

Fig. 1. Two programming levels

The behavioral level: at this level, programmer deals with features related to concurrency, persistence, distribution, and fault-tolerance. Such (behavioral) features are implemented within classes, called *behavioral classes*, of which instances are called *behavioral objects*. GARF provides two kinds of behavioral classes: *encapsulator classes* of which instances are used to wrap data objects by controlling the way they send and receive messages, and *mailer classes* of which instances are in charge of performing encapsulator's communications. GARF offers an extensible library of built-in behavioral classes that support remote communication, asynchronous invocation, concurrency control, persistence, active replication and passive replication (see section 3 below).

2.2 GARF objects

Data objects: these are passive entities that communicate in a *point to point, synchronous* request/reply manner. A request represents an operation invocation involving two objects: a client which invokes the operation and a server on which the operation is invoked. The reply (returned to the client) represents the result computed by the server when executing the invoked operation. A request or a reply communication is named *message sending* as in most object-oriented languages.

Encapsulators: these objects are used to wrap data objects by controlling how the latter send and receive messages. Each data object can be bound to an encapsulator. The encapsulator intercepts each request/reply communication (message), sent or received by its associated data object, in a transparent way[2]. So, every message sent *to* the data object by another data object passes through its encapsulator, as well as every message sent *by* the object. A data object and its encapsulator are always located on the same node. Hence, when an encapsulator is placed on a node, its associated data object is also placed on that node. According to its semantics, the encapsulator performs pre and post-actions which handle *asynchrony, concurrency control, persistence, replication,..*(section 3).

Mailers: these objects are used to perform (remote) communications between encapsulators. Each data object bound to an encapsulator, is also bound to a mailer class. A mailer of this class is created for each request sent to the encapsulator[3]. The mailer is created on the client node and is used to transmit the request to the encapsulator and to get back the reply. According to their semantics, mailers perform appropriate communications, e.g. *one to one, multicast, atomic multicast,..*(section 3).

Figure 2 depicts our basic communication model. It shows a (request/reply) communication between a client data object C, bound to an encapsulator $E(C)$, and a server data object S, bound to an encapsulator $E(S)$ and to a mailer class $M(S)$. The request sent by C is first intercepted by the encapsulator $E(C)$. The latter forwards the request to $E(S)$, through a mailer $Mi(S)$ (of class $M(S)$). The request arrives then at S which executes the required operation. The reply follows the reverse path, i.e., S, $E(S)$, $Mi(S)$, $E(C)$, and C. As we will see in the next section, the paths followed by request/reply communications provide data objects with various behavioral features, depending of the encapsulators and mailers that are involved.

[2] Data objects are not aware of encapsulators.

[3] There is a *one to one* correspondence between mailers and requests sent to the encapsulator.

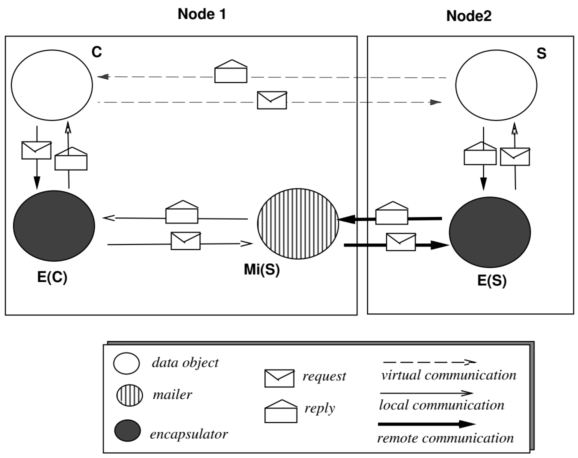

Fig. 2. The communication model

3 Distributed programming in GARF

Distributed programming in GARF consists in:

1. designing (or reusing) data classes,
2. binding data objects, either to existing encapsulator classes and mailer classes, or to newly designed ones;
3. placing encapsulators, and the data objects bound to them, on network nodes.

Every data class is derived from the built-in class *DataObject*. The class *Encapsulator* (resp. *Mailer*) is the root of the encapsulator (resp. mailer) class hierarchy. The class *Encapsulator* defines the basic operations `inRequest()` and `outRequest()`. The operation `inRequest()` is invoked before each invocation of a data object bound to an encapsulator, while the operation `outRequest()` is invoked after each invocation of the data object. The class *Mailer* defines the operation `sendRequest()` which is invoked by the client encapsulator's `outRequest()`. All these operations take as argument the invocation itself which is considered at this point as a first class object. The class *DataObject* defines the operation `garfNew()` which is performed at each data object creation. This operation can be refined in subclasses to bind a data object to an encapsulator and to a mailer class.

The classes *Mailer* and *Encapsulator* do not provide any specific behavioral features except remote communications support. In other words, data objects bound to basic encapsulators (class *Encapsulator*), and to the basic *Mailer* class communicate using a RPC scheme without any concurrency, persistence or fault-tolerance characteristic. By refining `inRequest()` and `outRequest()` in *Encapsulator* subclasses, and/or by refining `sendRequest()` in *Mailer* subclasses, one can implement various behaviors. GARF offers a built-in library compound of a hierarchy of encapsulator classes and a hierarchy of mailer classes (fig 3) that provide support for asynchrony, intra-object concurrency control, persistence and replication. In the following, we describe this support.

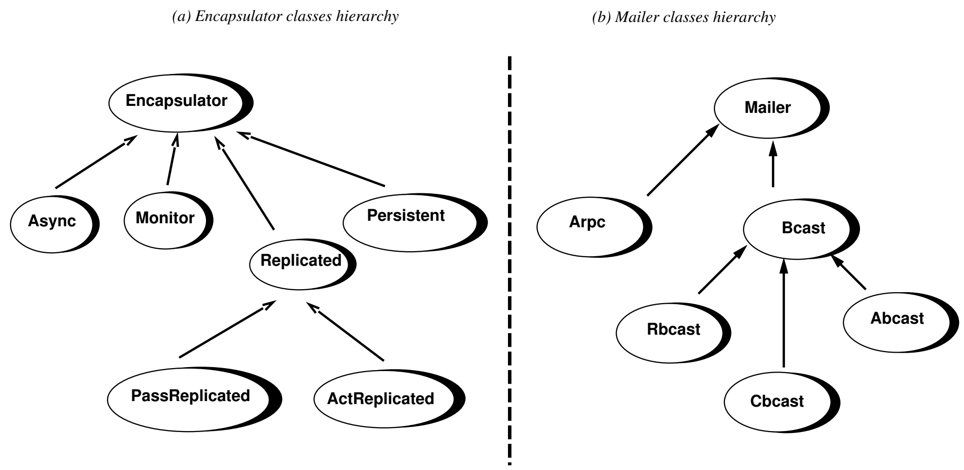

Fig. 3. Behavioral classes hierarchies

3.1 Asynchronous invocation

The synchronous request/reply communication (RPC) is based on a well understood abstraction (the procedure call). It is now a widely used communication facility in distributed systems [6,21]. Nevertheless, the client of such an invocation is always blocked during the execution of the remote operation. In many cases, it is preferable that the client turns its attention to another job in order to take advantage of the potential parallelism.

As in [23], we overcome this RPC limitation by introducing a *future* mechanism which implements the asynchronous invocation abstraction. The client of an asynchronous invocation receives immediately a future[4] without waiting for the actual result of the invoked operation. The client does not have to wait until it tries to read the value of the future: that is, until it until it invokes an operation on the future which value has not yet arrived. In this case, the client is blocked until the value (reply) arrives. Such a communication is particularly useful when the client and the server are located on different nodes since the communication delay is not negligible.

We enable two asynchronous behaviors:

Client asynchrony: asynchrony is introduced by the client's encapsulator. More precisely, when the client data object invokes an operation, its encapsulator creates a *thread* and returns a future (fig. 4). The thread is in charge of transmitting the invocation to the server's mailer, waiting for the reply and passing it to the future.

Server asynchrony: asynchrony is introduced by the server's mailers. When a mailer receives an invocation to forward, it creates a *thread* and returns a future (fig. 5). The thread is in charge of transmitting the invocation to the (remote) encapsulator, waiting for the reply and passing it to the future.

[4] A future is the container of the invocation result which may be not yet available.

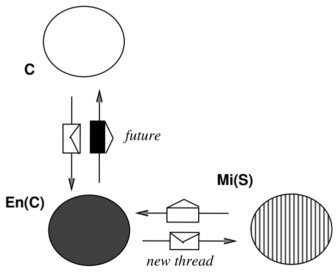

Fig. 4. Client asynchrony

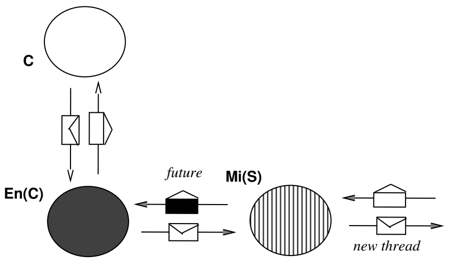

Fig. 5. Server asynchrony

We have designed an encapsulator class named *Async* which implements the client asynchrony and a mailer class named *Arpc* which implements the server asynchrony. One can provide a data object with either the first behavior by binding it to an encapsulator from the class *Async*, or the second behavior by binding it to the mailer class *Arpc*, or even both behaviors by binding it to an encapsulator from the class *Async* and to the mailer class *Arpc*.

3.2 Synchronization

Since data objects can be invoked concurrently, it is necessary to provide them with appropriate concurrency control that schedule incoming invocations. The simplest concurrency control[5] is the *monitor* like control. Such control ensures that only one invocation is treated at a time, i.e., it guarantees mutual exclusion of operation executions. While an operation is executing on the object, other invocations wishing to be treated must wait on an entry queue. When an operation terminates, the invocation at the top of the entry queue, if any, is treated, i.e., the corresponding operation is executed. As in [19], we introduce such behavior through encapsulators. More precisely, we provide an encapsulator class *Monitor*, which offers the mutual exclusion capability. An encapsulator from this class maintains an *invocation queue* in which it stores the invocations until their turn arrives.

A data object bound to an encapsulator from the class *Monitor* executes one operation at a time (fig. 6). It is obvious that such behavior might be too restrictive in many situations. One

[5] We are concerned here with intra-object concurrency control vs. inter-object concurrency control [12].

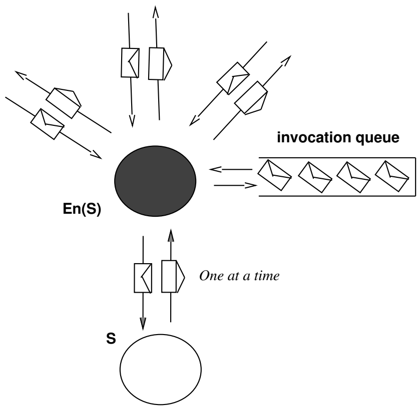

Fig. 6. Monitor like behavior

can refine the mutual exclusion policy in subclasses of *Monitor* and introduce intra-object concurrency (e.g. *readers/writer*).

3.3 Persistence

An object is persistent if it "dies at the right time" [16]. Generally, an object is said to be persistent if it does not disappear unless the programmer or user demands it explicitly. As a consequence, a persistent object can survive to the session that created it [2].

Supporting persistent objects requires global persistent names and stable storage. Persistent names are required to reference the object at any time and stable storage is required to preserve the object state between sessions. As for client asynchrony and synchronization, we provide persistence capabilities through encapsulators. We introduce a subclass of *Encapsulator* named *Persistent*. A data object bound to an encapsulator from this class is automatically made persistent. Such an encapsulator has a global persistent name, and a stable storage allocated to it. The encapsulator and its associated data objects are not deleted unless the programmer demands it explicitly by invoking the operation `die()` on the encapsulator.

An encapsulator from the class *Persistent* provides two operations `store()` and `retrieve()` which respectively store and retrieve the data objects' state to and from the stable storage (fig. 7). Hence, when a data object is created and bound to an encapsulator from the class *Persistent*, the encapsulator checks whether the data object has already been created within a previous session[6]; in this case it retrieves the data objects' state from the stable storage. After each operation execution, the encapsulator stores the objects' state into stable storage. This behavior can be refined in subclasses of *Persistent*, in order to be used within a transaction-based mechanism for example [11].

[6] The encapsulator uses its global persistent name for this purpose

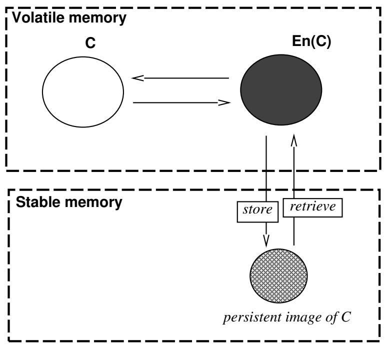

Fig. 7. To and from stable storage

3.4 Replication

Replication has been proven to be a powerful means to mask failures and increase availability [7]. Assuming *reliable communication channels* and *fail-silent nodes*, an object is reachable as long as it has a replica running on a non-faulty node.

Replication facilities in GARF are provided for data objects bound to encapsulators from subclasses of the class *Replicated*. Encapsulators from these classes can be gathered inside groups of which global names are maintained by *group managers*. A *group manager* is a particular GARF object used to store the member list of a group of encapsulators. When an encapsulator leaves the group, i.e., its node fails, the group manager removes its name from the list of members.

To replicate data objects of a class D, one must refine the operation `garfNew()` within the class D to create k copies of that object and bind them to k encapsulators of *Replicated* subclasses. The encapsulators (and thus their associated data objects) can then be located on failure independent nodes. To preserve the consistency of data object replicas, encapsulators must follow some agreements on how to forward invocations to their data objects. We have considered two kinds of agreements:

Active replication: every replica encapsulator receives the invocation and forwards it to its data object which treats it and returns a reply (fig. 8). There is no need here of inter-replicas coordination. That replication scheme is provided by the encapsulator class *ActReplica* and the mailer class *Abcast*. Mailers from the class *Abcast* transform an invocation into a reliable totally ordered multicast to the encapsulator replicas and waits for the first reply. Such a multicast guarantees that every replica treats an invocation or none, and that invocations are treated in the same order by all the replicas [5].

Passive replication: a single encapsulator called the *primary* replica receives the invocation, forwards it to its data object which treats it and returns the reply. At regular intervals, the primary replica sends checkpoints to other replicas which act as backups. If the primary replica fails, the backups elect a new primary. The class *PassReplica* implements encapsulators that behave that way. The mailer used at the client side can be simply created from

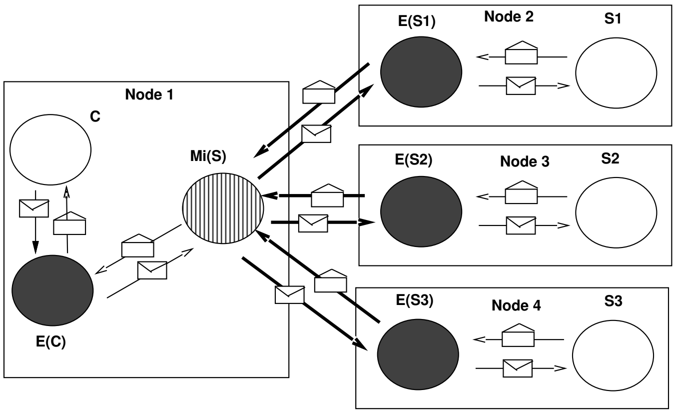

Fig. 8. Active replication

the basic class *Mailer*, since there is only a need for point to point communication between the client encapsulator and the primary encapsulator replica.

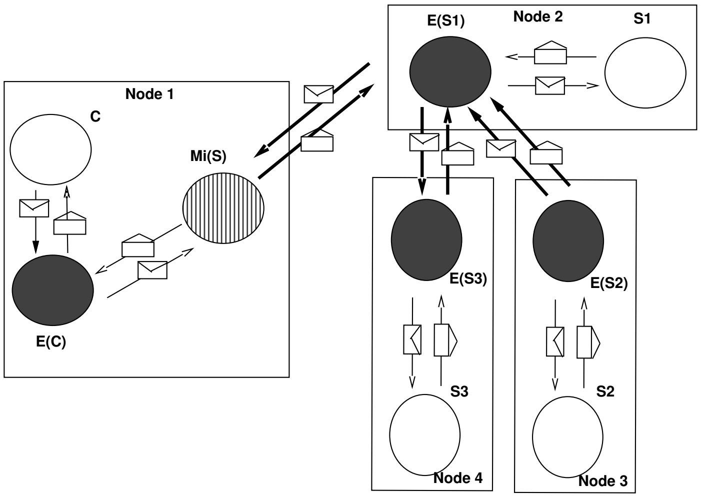

Fig. 9. Passive replication

4 Example: The Distributed Diary Manager

In this section, we illustrate our approach by describing the design of a fault tolerant distributed application built with the GARF system. The application is a distributed diary manager called the DDM. At the functional level, the application is designed in terms of data classes, assuming a volatile, centralized, and failure free environment. At the behavioral level, the programmer deals with features related to concurrency, persistence, distribution, and fault-tolerance, by using built-in encapsulator and mailer classes, or designing new ones.

4.1 Designing the DDM at the functional level

The DDM application is aimed to manage a diary for each user. A diary is associated to its owner through a diary index which manages a list of (user name, diary) pairs. A diary stores the list of meetings to which its owner participates. Each meeting is composed of a period and a list of participants. A diary service is the user-interface of DDM and enables to list all diary owners, to create or delete a diary, to plan or cancel a meeting with other owners and to list all the meetings a user is expected to attend.

At the functional level of the DDM, we designed mainly four data classes, derived from the basic abstract class *DataObject*: *DiaryIndex, Diary, Meeting*, and *DiaryService*. Class *DiaryIndex* implements operations returning a diary given a user name, listing the names of all diary owners and adding or removing a user. This class has a single instance that we will refer to as *the diary index* . Class *Diary* implements operations that allow to list all the meetings contained in a diary and to update the latter. This class has one instance per user. Class *Meeting* implements operations allowing to list its participants, to give the meeting period and to remove a user from the participant list. Each time a user plans a meeting, a new instance is created. Finally, class *DiaryService* describes the user-interface of DDM. An instance of this class is created each time a user starts to work with the application.

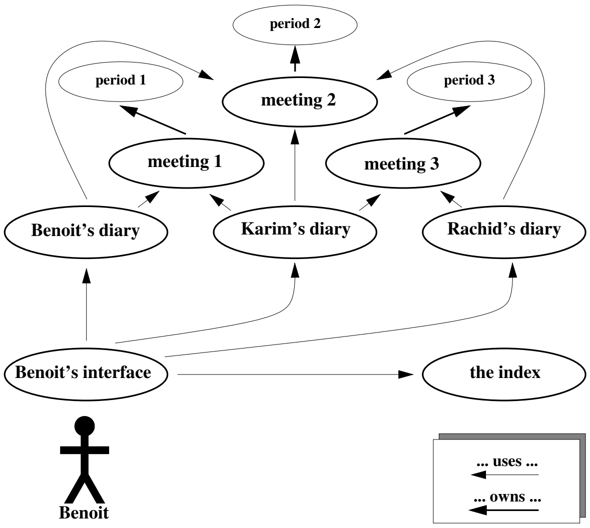

Fig. 10. Data objects of a DDM execution

Figure 10 shows data objects involved in a DDM execution. The figure points out the dependencies between instances of DDM classes. The *uses* relation expresses the client-server model in which a client object needs a server object to accomplish some services. The *owns* relation expresses a particular case of the *uses* relation, where the client is the unique user of the server object. Such relations allow to identify globally shared objects of the application. These objects are *diary services, index, diaries* and *meetings*; all other objects involved in DDM are local, e.g., *periods* and *owner names.*

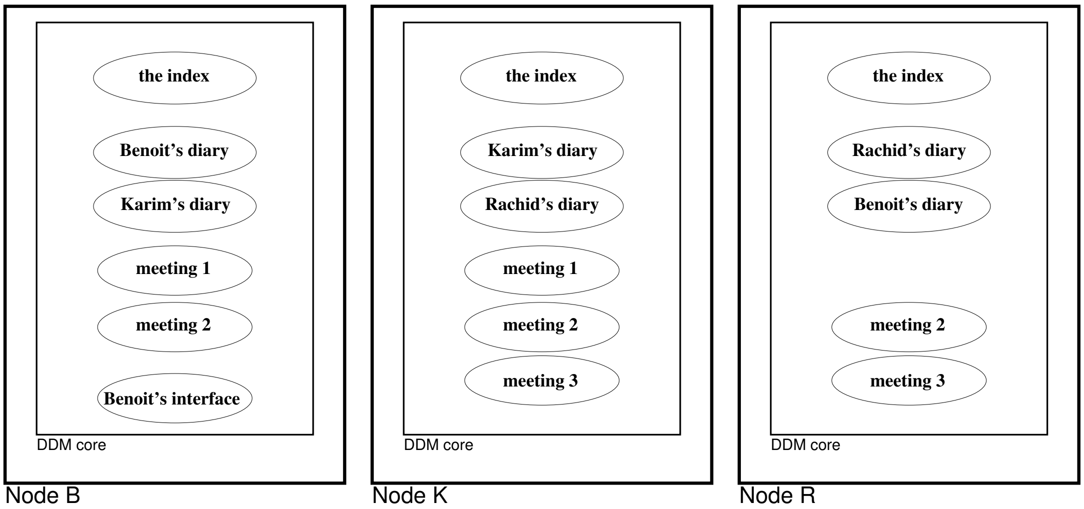

Fig. 11. Replication of DDM objects

4.2 Designing the DDM at the behavioral level

At the behavioral level, the design of DDM is extended with behavioral features, using encapsulator and mailer classes. Since DDM is intended to support several users at the same time, the diary index, diaries and meetings may be accessed concurrently. In addition, DDM is aimed to manage long-lived information, i.e., when a user terminates a session with the DDM, he expects to be able to reuse his diary in a further session. Finally, we require DDM to continue its execution despite a predefined number of node failures.

In our example, there are three potential users of DDM: Benoît, Karim and Rachid, each have a dedicated node, i.e., a workstation. Those nodes are named respectively B , K and R . A DDM core program is running on each of these three nodes, allowing to create DDM objects and to communicate with them at any time. When a user starts to work with DDM, a user-interface object (a diary service) is simply created on that node. When the user quits DDM, his diary service is deleted, while all the other objects that were created during his work, on any of the three nodes, keep on existing for later use.[7]

Fault-tolerance is achieved by replicating critical objects. An owned object can be seen as being a part of its owner (figure 10). With this in mind, we can now focus only on the replication of what we called the globally shared objects, i.e., the index, diaries, meetings, diary services. The diary index is a critical component. We want it to be replicated on each node where DDM is running. Furthermore it exists even though no user is registered. In our case, it means that a replica of the diary index is present on each of the three nodes, hence it can tolerate the failures of two nodes. A diary is always present on the node of its user, plus on one other node where DDM is running. The choice of that second node is made according to some policy of load balancing of objects. A diary can thus tolerate the failure of a single node. A Meeting's replication rate depends on the number of the meeting participants. Indeed each participant has a replica of all its meetings on his node. This way, even though all other nodes have failed, Benoît for example could still consult his list of

[7] The DDM core program can be seen as some kind of Unix daemon.

meetings. As said before, a diary service is the user-interface of DDM. It disappears when its user decides to stop working with DDM. Therefore it is not replicated at all and is always on the node of his user.

The difference of replication policies leads to several qualities of service depending on the operation the user wants to perform. For example listing owners and meetings is always still possible after two failures, while planning a meeting might not. It means that Karim could get the information that Rachid is registered, but could not be able to plan a meeting with him. To recover from failures, we consider persistence capabilities. Otherwise, if two nodes fail, Rachid for instance could well have irreversibly lost his diary. Furthermore, if all three nodes fail at the same time, DDM will be completely reinitialized and all information lost. In order to avoid loss of objects' states after failures, we make the diary index and all diaries and meetings persistent.

We now list the encapsulator and mailer classes used to achieve the various objects' behaviors specified above. The diary index, the diaries and meetings are shared and persistent objects. Their operations are performed in a *readers/writer* way, and their intermediate states are stored on stable memory. To support these features we designed an encapsulator class named *DDMEncapsulator*, which inherits its operations from classes *Monitor*, *Persistent*, and *Replicated*. Therefore, the diary index and each diary and meeting are bound to an encapsulator of class *DDMEncapsulator*. When a *DDMEncapsulator* instance is created, we specify both its replication rate and the locations of its replicas. We consider that strong consistency must always be insured for replicated objects, so we use a mailer class that provides totally ordered causal multicast: the *Abcast* mailer class.

5 Related work

5.1 Single level languages

Most of object-based systems designed for distributed applications provide a single programming level: both functional and behavioral features are described at the same level. This approach leads to a tradeoff between flexibility and modularity. Either behavioral features are defined in an uniform manner (reducing flexibility), or they can be modified but affecting functional aspects (reducing modularity).

Argus [15], Guide [14] and Arjuna [20] are well known examples of single level programming distributed systems. In all of these systems, objects communicate through Remote Procedure Call (RPC). As we have discussed in section 3, this is not always the desired behavior, and in many cases one might prefer an asynchronous style of communication. In addition, these systems provide transactions to maintain object consistency. In Arjuna and Guide, one has to write specific primitives that express transaction boundaries within object operations. This complicates the readability of operations and decreases modularity. Indeed, one cannot add transaction semantics to an operation without accessing it. By encapsulating all *Guardians* (large grain objects) invocations within transactions, the Argus system avoids this problem. However, as we pointed it out in [11,13], the systematic approach of nesting transactions does not always provide the best choice. In fact, there are many cases in which the requested operation does not need to lie within the client transaction boundary. A request involving a benevolent side effect, such as garbage collection, does not need to be designed as a subtransaction but as an independent top-level transaction.

5.2 Reflective systems

Apertos [22], and ABCL/R2 [18] are significative examples of object-based systems which provide powerful flexibility capabilities. These systems are said to be *reflective*. Broadly speaking, *reflection* is the process of reasoning about and acting upon the system itself [17]. In both Apertos and ABCL/R, each object is bound to a group of *meta-objects* describing the object behavior. For example, an object may be bound to a meta-object describing how the object communicates with other objects, a meta-object describing how to handle a faulty operation, and a meta-object describing how the virtual memory supporting the object is managed. Each meta-object is also bound to a group of meta-objects, and so on. This leads to a meta-tower for each object.

Although flexible, such systems have complicated semantics and may bring unnecessary additional complexity. We believe that since the only way to access to an object is to invoke one of its operations, it suffices to control the invocation process in order to express various behaviors. Hence, our encapsulator and mailer model may be viewed as a simpler reflective approach.

5.3 First class interactions

The most similar approaches to ours are those providing first class interaction supports. These are systems like SOS [8], XENOOPS [3] and Sina [1]. According to these approaches, object interactions are seen as first class citizens, i.e., at the same level as other objects.

Both XENOOPS and SOS offer some kind of flexible proxies that allow generic mechanisms for identification, location, storage, replication and concurrency control. In XENOOPS, each object is invoked through a *reference*, which is a particular object that determines the required action to be performed before the actual invocation. Nevertheless, since programmers have only control on what happens before an invocation, and never after, it is not possible to program post-actions following an operation execution, nor to control what happens on the client side when it invokes an operation. For example, it is not clear how one may compute selective *synchronous/asynchronous* invocations without accessing the client operation's code.

In SOS, distributed applications are made of *fragmented* objects. A fragmented object is a group of elementary objects which can be located at several address spaces. From its client point of view, a fragmented object appears as a single local object accessible through its exported interface. From its designer point of view, a fragmented object is seen as a collection of elementary object located at a different address spaces. Nevertheless, the proposed approach does not allow a modular distinction between functional and behavioral aspects, which are both mixed inside fragmented objects' implementations.

The model proposed in the SINA system is called the *composition-filter model*. Each object is associated to a set of *incoming filters* and *outcoming filters*. An incoming filter is performed when an object receives a request and an outcoming filter is performed when an object sends out a request. Filters may delay a message, redirect it of change the message. Since filters are locally associated to objects, programmers may only act on what happen when the message arrives at the object side. Hence it is impossible to compute a replicated object, in a transparent way from clients. More precisely, there is no way to multicast an invocation to an object, in a transparent way to its client.

6 Concluding remarks

This paper describes the computational model of GARF: an object-oriented distributed system aimed to enhance modularity and flexibility of applications. The GARF model is based on two programming levels: a functional level and a behavioral level. At the functional level, programmers focus on software functionalities, and at the behavioral level, they address concurrency, persistence, distribution and fault tolerance features. Such features are programmed within specific classes of which instances are encapsulators and mailers. These classes implements appropriate behavioral abstractions. In the paper we described some of the built-in classes provided by the GARF system which support client asynchrony, server asynchrony, monitor like concurrency control, persistence, active replication, and passive replication.

The incremental way of programming promoted by the GARF approach enhances both modularity and flexibility. Modularity is enhanced since different aspects of a same application are implemented inside independent classes. Data classes that implement functional aspects and encapsulator and mailer classes which implement behavioral features. One can modify the implementation of a behavioral class, e.g. a multicast class, without affecting the rest of the application. Flexibility is enhanced since the built-in behavioral abstractions can be refined to achieve the desirable efficiency according to application's semantics.

A GARF prototype has been implemented on top of Smalltalk [10] and the ISIS [4] toolkit for fault tolerant distributed programming. The GARF approach has been applied to a simple fault-tolerant application (see [9]), and is now applied to the design of more complex application in order to refine the model and to extend the mailer and encapsulator hierarchies.

References

1. M. Aksit, K. Wakita, J. Bosh, L. Bergmans and A.Yonezawa. Abstracting Inter-Object Communications Using Composition Filters. *Internal Report 0924-3755, Univ. of Twente.* 1987.
2. M.P. Atkinson, P. Bailey, W. Cockshott, K. Chisholm and P. Morrison. Procedures as Persistent Data Objects. In *ACM Transactions on Programming Languages and Systems.* 1985.
3. S. Bijnens, W. Joosen and P. Verbaeten. A Reflective Invocation Scheme to Realise Advanced Object Management. In R. Guerraoui, O. Nierstrasz and M. Riveill editors, *Object Based Distributed Programming*, Springer Verlag publisher. 1994.
4. K. Birman, T.A. Joseph and F. Schmuck - ISIS: A Distributed Programming Environnment, User's Guide and Reference Manual. *Internal Report, Dept of Computer Science, Cornell Univ.* 1991.
5. K. Birman, A.Schiper and P.Stephenson. Lightweight causal and atomic group multicast. In *ACM Transactions on Computer Systems.* August 1991.
6. D. Cherriton. The V Distributed System. In *Communications of the ACM.* March 1988.

7. F. Cristian. Understanding Fault-Tolerant Distributed Systems. In *Communications of the ACM.* February 1991.
8. P. Dickman and M. Makpangou. A Refinement of the Fragmented Object Model. In *IEEE Proceedings of IWOOS'92: the International Workshop on Object Orientation in Operating Systems.* 1992.
9. B. Garbinato, R. Guerraoui and K. Mazouni. Programming Fault-Tolerant Applications Using Two Orthogonal Object Levels. In *Proceedings of ISCIS'93: the International Symposium on Computer and Information Science.* 1993.
10. A.J Goldberg and A.D Robson. SMALLTALK-80: The Language and its Implementation. Addison Wesley publisher. 1983.
11. R. Guerraoui, R. Capobianchi, A. Lanusse and P. Roux. Nesting Actions through Asynchronous Message Passing: the ACS protocol - In *Proceedings of ECOOP'92: the European Conference on Object Oriented Programming,* Springer Verlag publisher. 1992.
12. R. Guerraoui. Towards Modular Concurrency Control for Distributed Object Oriented Systems. In *IEEE Proceedings of FTDCS'93: the International Workshop on Future Trends in Distributed Computing Systems.* 1993.
13. R. Guerraoui. Nested Transactions: Reviewing the Coherency Contract. In *Proceedings of ISCIS'93: the International Symposium on Computer and Information Science.* 1993.
14. S. Krakowiak, M. Meysembourg, V.H Nguyen, M. Riveill and C. Roisin. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. In *Journal of Object-Oriented Programming.* September/October 1990.
15. B. Liskov and R. Sheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. In *ACM Proceedings of SOPL'82: the Symposium on Principles of Programming Languages.* 1982.
16. C. Low. A Shared Persistent Object Store. In *Proceedings of ECOOP'88: the European Conference on Object Oriented Programming,* Springer Verlag publisher. 1988.
17. P.Maes. Issues in Computational Reflection. In P. Maes and D. Nardi editors, *Meta-Level Architectures and Reflection*, North Holland publisher. 1988.
18. S. Matsuoka, T. Watanabe and A. Yonezawa. Hybrid Group Reflective Architectures for Object-Oriented Concurrent Reflective Programming. In *Proceedings of ECOOP'91: the European Conference on Object Oriented Programming,* Springer Verlag publisher. 1991.
19. G.A Pascoe. Encapsulators: A New Software Paradigm in Smalltalk-80. In *ACM Proceedings of OOPSLA'86: the Conference on Object-Oriented Programming Systems, Languages and Applications.* 1986.
20. G.D Parington and S. Shrivastava. Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems - In *Proceedings of ECOOP'88: the European Conference on Object Oriented Programming,* Springer Verlag publisher. 1988.
21. R. Schantz, R. Thomas and G. Bono. The architecture of the Cronus Distributed Operating System - In *IEEE Proceedings of ICDCS'86: the International Conference on Distributed Computing Systems.* 1986.
22. Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *ACM Proceedings of OOPSLA'92: the Conference on Object-Oriented Programming Systems, Languages and Applications.* 1992.
23. E. Walker, R. Floyd and P. Neves. Asynchronous Remote Operation Execution in Distributed Systems. In *IEEE Proceedings of ICDCS'90: the International Conference on Distributed Computing Systems.* 1990.

This article was processed using the LaTeX macro package with LLNCS style