Lessons from Designing and Implementing GARF

Rachid Guerraoui Benoît Garbinato Karim Mazouni

Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract. GARF is an object oriented system aimed to support the design and the programming of reliable distributed applications on top of a network of workstations. The specificity of GARF resides in its incremental programming model, and its extensible library of generic components. In this paper, we first give an overview of GARF. Then we assess its features with respect to its programming model, its abstraction library, and the implementation of its prototype. We finally present the perspectives of future design and implementation.

1 Introduction

1.1 Distribution and reliability

Programming distributed reliable applications is a hard task. The complexity of such applications is due to their *behavioral* features, i.e. concurrency expression and concurrency control, distribution and replication.

Several reliable distributed toolkits provide primitives that deal with such behavioral features. Isis [2], Psync [19], Transis [1] and Phoenix [16] are examples of such toolkits. Although powerful, these toolkits are at a too low level (Unix process level), and thus very difficult to use for programmers without a broad experience in programming distributed and reliable applications. Systems like Arjuna [14] or Electra [15] support a higher programming level (object level) and provide adequate primitives for remote invocation, object replication and object group invocations. Since these primitives must nevertheless be mixed with application *functional* features (i.e. sequential and centralized aspects), they hamper extensibility and modularity. For example, an application written in a centralized context cannot be directly reused for a distributed and reliable purpose. Furthermore, as stated in [20], adding new abstractions for reliable and distributed programming cannot be done in a straightforward way.

1.2 GARF Characteristics

GARF is an object-oriented system aimed to support the design and programming of reliable distributed applications, at a higher level than existing toolkits such as Isis, Psync, Transis and Phoenix. More precisely, GARF can be viewed as an intermediate between these toolkits and the application. The first prototype of GARF is actually based on Isis, and a new prototype is currently under developement on top of Phoenix. In comparison to systems like Arjuna or Electra, GARF promotes software modularity by clearly separating the behavioral features that concern concurrency, distribution and reliability, from functional ones than concern traditional sequential and centralized aspects.

GARF encourages an incremental programming methodology. First, the GARF programmer may design and implement application components in a centralized environment, focusing only on their functionalities. In further steps, without modifying the previously written code, the programmer may turn to behavioral features, by expressing and controlling concurrency, distributing the application over a network, and replicating its critical components to increase the reliability. Of course, all these steps, including the first one, may be performed concurrently, but the code written for different steps is separated.

GARF offers an extensible *behavioral* library of abstractions for various kinds of concurrency controls, remote object invocations, e.g *synchronous/asynchronous*, as well as for object group invocations, e.g *reliable/fifo/causal/atomic*. These abstractions enable to support for example both *passive* and *active* object replication [3]. GARF abstractions can be chosen according to the application semantics. An experienced programmer can implement new behavioral abstractions by combining existing abstractions, or even by using low level mechanisms provided by the *system* library of GARF. The current prototype of GARF has been implemented in Smalltalk on top of Isis. This prototype was used for the development of a reliable distributed diary manager on a network of Sun workstations [5].

1.3 Assessment and perspectives

The aim of this paper is to give a global overview of GARF, and then to present some feedbacks on its design and implementation. Among the lessons learned are:

- *Static object/behavior association is enough:* in the development of "classical closed" applications in a distributed context, there is no need for changing the behavior of objects at run-time.

- *One meta level is enough:* GARF introduce *encapsulators* and *mailers*, which can be viewed as *meta objects* [13], dedicated to describe object *communications*. The need for *meta objects* describing additionnal features such as ressource management was not identified, and the utility for *meta meta objects*, which describe the behavior of meta objects, is not obvious at all.

- *The asymmetric proxy model is too restrictive:* when considering replicated objects that can act as clients and servers in the same time, the classical proxy model (used in GARF) may lead to a proliferation of useless invocations, and even to inconsistencies in the case of updates. We discuss this issue and show how a symmetric model is more adequate in this case.

- *Behavioral classes are not independent:* the independence of the behavioral classes offered by GARF is not straightforward, and several pairs of classes are strongly coupled. This is actually not surprising as each behavioral class represents a specific abstraction for distributed programming, and these abstractions are rarely orthogonal.

- *Integrating transactions requires a flexible underlying toolkit:* the current GARF prototype is based on the Isis toolkit [2]. Although this toolkit has facilitated the implementation of GARF, the fact that it does not support transactions makes it difficult to maintain the consistency of several interrelated groups of object replicas. This situation occurs for example in the distributed diary manager application written with GARF. Porting GARF on top of Phoenix [11] will fortunately enable to circumvent this limitation.

1.4 Paper overview

We first recall the GARF programming model in Section 2. To illustrate the model, we briefly describe a reliable distributed application written in GARF: the reliable distributed diary manager. Then we give an overview of GARF architecture and implementation in Sections 3 and 4. The assessment of GARF is presented in Section 5. This assessment was done with respect to (1) the programming model, (2) the library of built-in abstractions, and (3) the current implementation. According to this assessment, we present some perspectives on GARF future development in Section 6. We finally summarize the GARF principal features and some future development.

2 GARF Programming model

In this section, we present the GARF programming model, which we illustrate through an application written in GARF: the reliable distributed diary manager.

2.1 GARF objects

GARF handles three kinds of objects: *data* objects, *behavioral* objects and *system* objects.

1. **Data objects** are passive entities that communicate in a *point-to-point, synchronous, request/reply* manner. A request is an operation invocation. It involves 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.

2. **Behavioral objects** can be viewed as *"meta data objects"* [13], used to describe behavioral features (i.e. concurrency, distribution, and reliability) of data objects. GARF supports two kinds of behavioral objects, *encapsulators* and *mailers*.
   - **Encapsulators** are used to *wrap* data objects by controlling the way they treat incoming and outcoming requests. GARF provides a library of encapsulator classes (Section 3). For example, the class `Replica` enables to create multiple replicas of a data object, in order to increase its availability.
   - **Mailers** are used to perform (remote) communications between encapsulators. GARF provides a library of mailer classes (Section 3). For example, the mailer class `Abcast` ensures that all replicas of an encapsulator receive concurrent requests in the same order.

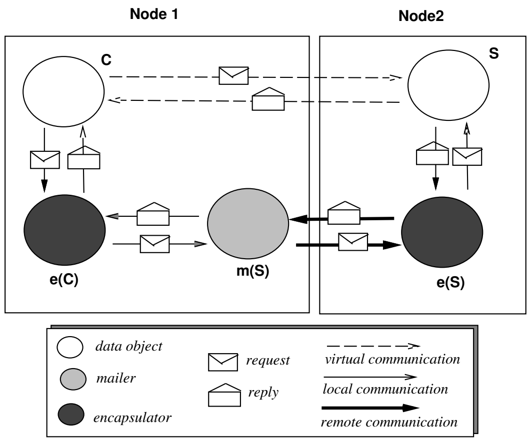

Fig. 1. data objects and behavioral objects

Figure 1 describes the runtime relation between data objects and behavioral objects. There are two data objects, `C` and `S`, two encapsulators `e(C)` and `e(S)`, and a mailer `m(S)` that handles the communications between the encapsulators.

3. **System objects** implement the low level mechanisms on which encapsulator and mailer classes are based. For example, the system class `Group` is used by the encapsulator class `Replica`.

2.2 Programming levels

The three kinds of objects supported by GARF convey three corresponding programming levels: (1) the *data level*, (2) the *binding level* and (3) the *system level*.

1. At the **data level**, the programmer describes, in terms of data classes (classes of data objects), functional aspects of the application in a centralized environment. At this level, the programmer needs not be familiar with concurrent, distributed, or reliable programming.

2. At the **binding level**, the programmer turns to concurrency, distribution, and reliability by binding data objects to adequate behavioral objects (i.e encapsulators and mailers). At this level, the programmer has to be familiar with the meanings of concurrent, distributed, and reliable abstractions. *Behavioral* classes are chosen according to the semantics of the application (see Section 2.3).

3. At the **behavioral level**, the programmer extends the library of behavioral classes. The programmer either combines existing behavioral classes to derive new ones, or uses *system objects* [6, 7]. At this level, the programmer must be familiar with the GARF system, and even with the underlying toolkit in case he tries to define new system classes. Note that this task is only necessary if the built-in behavioral library does not contain the desired abstractions.

2.3 Example: the reliable distributed diary manager

We illustrate the GARF programming model by describing the design of a reliable distributed application implemented with GARF. We give here a brief description of the application. More details can be found in [7]. We first present an overview of the application, then we sketch the functional and binding level. There was no need for programming at the behavioral level for this application.

Overview: The aim of the application is to handle the diaries of a group of persons, by enabling the automatic planning and cancellation of meetings. A diary is associated to its owner through a diary index which manages a list of user name and 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 a user-interface. It 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. The required reliability feature of the application is that, as long as its machine is up, a user must always be able (1) to consult its diary, and (2) to schedule a meeting with every other user of which machine is also up[1].

Functional level: At the functional level, we designed mainly four data classes: `DiaryIndex, Diary, Meeting`, and `DiaryService`. Class `DiaryIndex` implements operations returning a diary given a user name, listing the names of all

[1] We do not assume link failures here.

diary owners, and adding or removing a user. This class has one instance: 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. Note that an instance of `Meeting` is shared by all the participants in the meeting. Finally, class `DiaryService` describes the user-interface. An instance of this class is created each time a user starts to work with the application.

Binding level: At this level, the design of the application is extended with behavioral features, using encapsulator and mailer classes.

- **Behavioral classes:** Since the application is intended to support several users at the same time, the `diary index`, `diaries` and `meetings` may be accessed concurrently.
  1. The `diary index` is a critical component. We want it to be replicated on each node where the application is running. Furthermore, the `diary index` must exist even though no user is registered.
  2. A `diary` is always present on the node of its user.
  3. The replication rate of a `meeting` depends on the number of the participants in the `meeting`. Indeed each participant has a replica of all its `meetings` on his node. This way, even though all other nodes have crashed, on can still consult its meetings through its diary.
  4. As said before, a `diary service` is the user-interface. It disappears when its user decides to stop working. Therefore it is not replicated at all and is always on the node of his user.

  The encapsulator class which provides the desired behavior for replication is the class `ActiveReplica`: it ensures that all replicas treat an invocation, and return a reply. As long as one replica is alive, a reply is returned. The mailer class `Abcast` ensures that all the replicas are invoked in the same order, and hence ensures strong consistency. Furthermore, a mailer of this class selects one reply among all (the first one to arrive), and forwards it to the encapsulator.

- **Binding behavioral objects to data objects:** Data objects of a class `D` are bound to encapsulators and mailer classes at runtime, during the execution of the specific operation `garfNew:` on `D`. This operation is inherited by every data class and invoked whenever a data object is created. By default, the `garfNew:` operation binds each data object to an encapsulator of the basic class `Encapsulator`, and to the mailer class `Mailer`. These classes only forward local communication and do not perform any particular treatment. Hence, the main programming task at the binding level consists of modifying the `garfNew:` implementation, which is done in three steps (see figure 2):
  1. Creation of an encapsulator at the specified node.

2. Creation of the data object and binding the encapsulator to it [2].
3. Creating a mailer and binding the encapsulator [3] to it.

```
          garfNew: aCreationMessage
          ...
(step 1)  encaps := ActiveReplica groupOn: Set new add: 'lsesun1' add: 'lsesun2'.
(step 2)  encaps buildAndBind: self sending: aCreationMessage
(step 3)  mailer:= Abcast to: encaps.
          ...
```

Fig. 2. A `garfNew:` operation

Figure 2 shows an example of a `garfNew:` operation, which actively replicates the `diary index` on two nodes: 'lsesun1' and 'lsesun2'. The (strong) consistency of the replicas is maintained thanks to the `Abcast` mailer class, which ensures that concurrent requests are received in the same order.

The first step consists in creating an encapsulator of the class `ActiveReplica` and replicating it on two nodes 'lsesun1' and 'lsesun2'. The second step consists in creating the data object and binding the encapsulator to it. The third step consists in binding the encapsulator to the mailer class `Abcast`. The result of the `garfNew:` operation is shown in Figure 4.

3 GARF Architecture

GARF is not intended to be used in a stand-alone fashion, but rather on top of a reliable distributed toolkit, which provides low level primitives for remote communication, group management, and reliable multicast communication. The first GARF prototype has been implemented in Smalltalk, on top of the Isis toolkit. The GARF/Isis interface is written in C (Figure 3).

The GARF system is composed of a runtime and three class libraries: the *encapsulator* library, the *mailer* library, and the *system* library:

3.1 GARF run-time

The GARF runtime manages centralized object creation and communication, as well as associations between data objects and behavioral objects, specified by programmers. GARF runtime handles these associations by intercepting data object creation and communication.

[2] The data object is created on the encapsulator's node.
[3] And hence the data object.

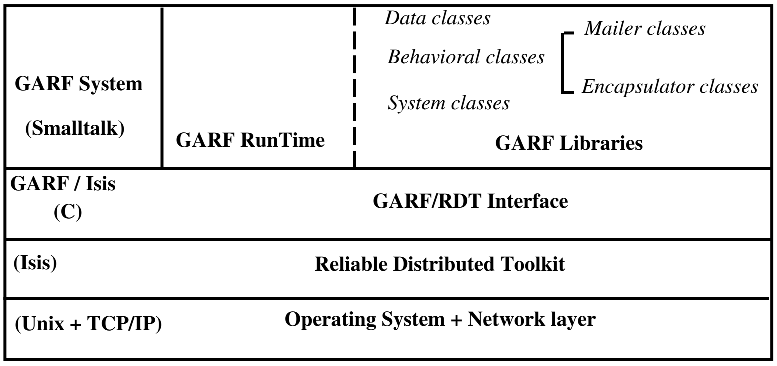

Fig. 3. GARF software layers

Each data object `S` is bound to an encapsulator (`e(S)`), and to a mailer (`m1(s)`) (see Figure 1). The data object and its encapsulator, i.e. `S` and `e(S)`, are always located at the same node. The runtime redirects each communication to and from `S` through `e(S)`. The encapsulator can thus control the behavioral features of the data object such as concurrency and replication. A clone `m(S)` of the mailer `m1(S)` is created whenever `S` is invoked by an object `C`. The mailer `m(S)` is located on the node of `C` and acts as a proxy of `e(S)`: `m(S)` forwards the invocation to `e(S)`. In Figure 4, `S` is bound to an encapsulator of the class `ActiveReplica` and to a mailer of the class `Abcast`. As shown in the figure, a mailer of the class `Abcast` transforms a *point-to-point* invocation into a reliable totally ordered multicast on a group of replicas. The replicas in the figures are `S1` and `S2`, respectively attached to the encapsulators `e(S)1` and `e(S)2`.

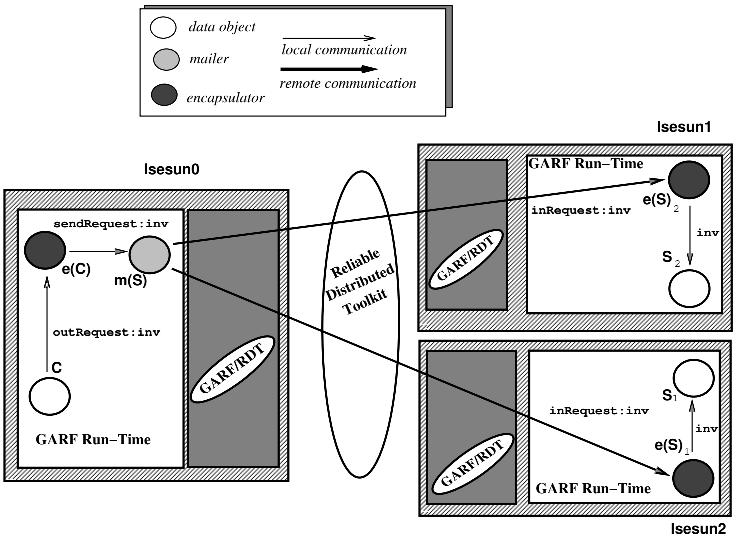

Fig. 4. GARF execution model

The GARF execution model can be described more accurately as follows:

- The GARF runtime intercepts each invocation `inv` from a data object client `C` on a data object server `S`, and *transforms* it into the invocation of the operation `outRequest:` on `e(C)`, with `inv` as an argument. The operation `outRequest:` is inherited from the basic class `Encapsulator` and redefined in the subclasses presented in Figure 5.
- `e(C)` executes operation `outRequest:` which invokes the operation `sendRequest:` on `m(S)` (with `inv` as argument). The operation `sendRequest:` is inherited from the basic class `Mailer` and can be redefined in the subclasses presented in Figure 5 (b).
- `m(S)` executes operation `sendRequest:` which forwards the invocation (possibly through a multicast as shown on Figure 4) to `e(S)` by invoking operation `inRequest:` (with `inv` as an argument) on it. Operation `inRequest:` is inherited from the basic class `Encapsulator` and refined in the subclasses presented in Figure 5 (b).
- `e(S)` executes the operation `inRequest:` and finally transmits the invocation `inv` to `S`.
- The reply follows the reverse *route*.

3.2 GARF abstractions

Encapsulator library: encapsulator classes are subclasses of the basic class `Encapsulator`, which provides two operations: `inRequest` and `outRequest`. These operations handle the way data objects receive and send requests. Defining encapsulator classes consists of programming these operations inside subclasses of `Encapsulator`. GARF provides a library of encapsulator classes (Figure 5 (a)) that handle concurrency expression (class `Active`), concurrency control (classes `Mutex` and `ReadersWriter`), and replication (classes `PassiveReplica` (passive replication), and `ActiveReplica` (active replication)). These classes are also detailed in [6] and [7].

Mailer library: mailer classes are subclasses of the basic class `Mailer` which provides the operation `sendRequest`. This operation handles the way communications are performed between encapsulators. Defining mailer classes consists of programming this operation inside subclasses of `Mailer`. GARF provides a library of mailer classes (Figure 5 (b)), which handle synchronous and asynchronous remote invocations (classes `Srpc`, and `Arpc`, as well as various models of multicasts (classes `Mcast` (reliable multicast), `Fbcast` (*fifo* reliable multicast), `Cbcast` (causal reliable multicast [2]), and `Abcast` (atomic reliable multicast [2]). These classes are also detailed in [6], [7], and [10].

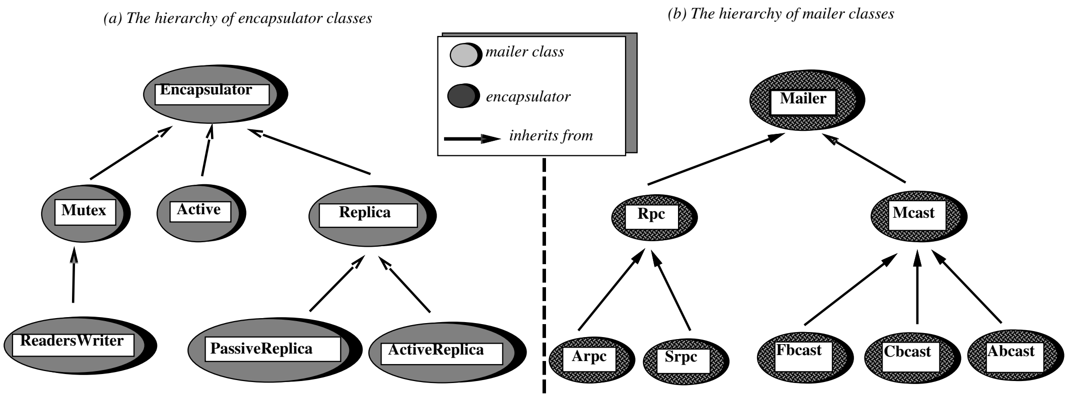

Fig. 5. GARF encapsulator and mailer classes

System library: this library contains a set of classes called *system* classes, of which instances are the *system* objects. These objects represent the main abstractions provided by the underlying toolkit. In our current prototype, the underlying toolkit is Isis, and system classes provided by GARF are for example: class `Group` for object group management, class `Future` for asynchronous invocation handling, and classes `ClassName` and `ObjectName` for invocation interception (as discussed in Section 4.1).

4 GARF Implementation

4.1 GARF runtime

The GARF runtime is in charge of centralized object management (i.e. object creation and communication), as well as associations between data objects and behavioral objects. Since the GARF runtime is an extension of the Smalltalk runtime, local object management is done by the latter.

Associations between data objects and behavioral objects are handled during data object creation and communication. The GARF runtime intercepts each invocation `new:` on a data class `D`, and transforms it into a `garfNew:` invocation on `D`. The runtime also intercepts each invocation from a data object `d`, and transforms it into an `outRequest:` invocation on `d`'s encapsulator. These interceptions are supported by two classes that we have implemented: `ClassName` and `ObjectName`.

- `ClassName`: during the initialisation phase of a GARF application, all data classes are substituted, within the Smalltalk `Dictionary`, by instances of the class `ClassName`. Hence, whenever the `new:` operation is invoked on a data class `D`, `new:` is actually invoked on the corresponding `ClassName` instance `cn(D)`. Since `cn(D)` does not provide the operation `new:`, the Smalltalk runtime raises an exception by invoking on `cn(D)` the operation `doesNotUnderstand:`, which we have implemented within the class `ClassName`, in such a way that it invokes the operation `garfNew:` on the data class `D`.
- `ObjectName`: the `garfNew:` operation performs the actual `new:`, to create a data object `d`, but it does not return `d`'s reference. It rather returns an instance `n(d)` of the class `ObjectName`, which owns the reference to `d`. Hence, when an operation is invoked on `d` by a client `c`, it is actually invoked on `n(d)`. Since `n(d)` does not provide such an operation, the Smalltalk-80 runtime raises an exception by invoking on `n(d)` the operation `doesNotUnderstand:`, which we have implemented within the class `ObjectName`, in such a way that it invokes the operation `outRequest:` on `c`'s encapsulator.

4.2 GARF abstractions

In the following, we describe the behavioral classes presented in Figure 5, and the system classes used for their implementation.

- **Encapsulator classes:** the implementation of these classes consisted in redefining the operations `inRequest:` and `outRequest:` inherited from the basic class `Encapsulator`. To implement the classes `Mutex` and `ReadersWriter` (both classes deal with concurrency control), we have redefined the operation `inRequest:`, so as to perform every operation of the associated data object in a *critical section* (with a well known semantic in the case of the `ReadersWriter` class). This implementation uses objects of the system class `Semaphore` provided by Smalltalk. To implement the class `Active:` (which offers *client asynchrony*), we redefined the operation `outRequest:`, in such a way that it creates a dedicated process for each invocation from the associated data object. The process is used to wait for the reply. This behavior enables the client to continue its execution after the invocation, and to obtain the reply at a later stage (using a *future* object). The implementation uses the system class `Process` provided by the Smalltalk environment, and the system class `Future` that we have implemented [6]. Neither operation `inRequest:` nor `outRequest:` are redefined in the class `ActiveReplica`. Indeed, an encapsulator of this class only forwards invocations and replies, to and from its associated data object: an actively replicated object acts as if it was not replicated [3]. The operation `inRequest:` has been redefined in the class `PassReplicated`, so that only one replica (the oldest created) treats an invocation, returns the reply, and then sends its state to other replicas.

- **Mailer classes:** Mailers act as proxies of encapsulators (Figure 4). In the current implementation of GARF, every encapsulator is represented by a *group* which corresponds to a set (possibly with a single element) of identical copies of an object. A group abstraction is implemented by an instance of the system class `Group`. To benefit from group membership monitoring and communication primitives provided by Isis, we have directly mapped GARF encapsulator groups on Isis process groups. The GARF/Isis interface is a set of C routines called within Isis threads, for each communication between some mailer and its corresponding encapsulator (through operation `sendRequest:`). Every encapsulator invocation (through its group name) by a mailer is translated by the GARF/Isis interface into an Isis multicast. Object communications are coded and decoded using the BOSS (*Binary Object Streams Services* classes provided by Smalltalk.
  To avoid interference between Smalltalk-80 processes and Isis threads, we implemented the GARF/Isis interface inside a dedicated Unix process.
  The classes `Arpc` *(server asynchrony)* and `Srpc` *(server synchrony)* are used for non-replicated encapsulator invocation. Both classes use however the class `Group`, with the special case of a single element in the group. The class `Arpc` uses in addition the classes `Process` and `Future` (to handle asynchrony). All `Mcast` subclasses use class `Process`, and each of these classes represents a specific multicast semantics supported by Isis.

4.3 Performances

Our measurement was carried out using a Sun SPARCstation equipped with 32 Mbytes RAM and running Solaris 2.2. The measurement took place on normal workday, so the workstation had medium to high load: it was running XWindows as well as interactive applications (emacs, mosaic, etc.).

The GARF runtime introduces an overhead factor of 70 without considering distribution and reliability. Indeed, a normal (neither encapsulator nor mailer are associated to objects) Smalltalk invocation of an object has a response time of $9\mu s$. When intercepted by the GARF runtime and redirected to mailers and encapsulators, the read invocation takes $628\mu s$. The reasons for such a high factor have to do with how Smalltalk virtual machine executes and how it handles exceptions. During normal execution, the virtual machine is partially bypassed and object operations directly execute on the physical processor. When an exception is raised however, the virtual machine reinterprets the operation that caused the exception, which makes the execution much slower. This is what happens with the operation `doesNotUnderstand:`.

The time spent in the GARF runtime and in data objects is however small compared to $251ms$ a read invocation takes when objects are distributed and replicated. This test scenario was based on the example introduced in Figure 4, i.e. some client object interacts with a server object that is replicated twice. Using `ActiveReplica` encapsulators and `Abcast` mailers, the client invokes a read operation on the replicated server and receives an integer managed by the server as a reply.

5 Assessment

In this section, we assess the main features of GARF, considering (1) its programming model, (2) its abstraction libraries, and (3) the implementation of its prototype. The assessment is based on several developments done with GARF in extending its abstraction libraries, on the implementation of the reliable distributed diary manager, and on the performance measures. This assessment must not be viewed as a general assessment on the design and implementation of object based distributed systems.

5.1 Programming model

Incremental programming is nice: One of the fundamental characteristics of GARF is its incremental way of programming. One can focus on different aspects of an application at different stages of the development. This incremental approach was very convenient in the design and implementation of the distributed diary manager application. It was very helpful, first to test the sequential aspects, second to consider concurrency, and then to deal with distribution and reliability (i.e replication). The code written in each step has never been modified at a later step.

There are applications for which this incremental way of programming can however not be easily applied. For example, several applications are intrinsically concurrent and it is not possible to separate their sequential features from their concurrent behaviors. There are also cases where taking into account the distribution aspect, at an early stage in the object design, can lead to better performances. We believe that, in these situations, even if different aspects are treated in the same time, modularity and reusability are increased by separating the code related to each aspect.

Static association is enough: The GARF programming model is based on the associations between data objects and behavioral objects. These associations are performed at runtime (i.e dynamically), which enable for example to change the encapsulator associated to a data object. We believe however that such a feature is not necessary and these associations could be done in a static way, and once for all. Applications that would require a dynamic association scheme, are those that necessitate dynamic change of behavior. These "open" applications, such as operating systems, are actually out of the scope of GARF.

One meta level is enough: Encapsulators and mailers can actually be viewed as meta data objects since they describe how data objects behave [21]. The GARF model can thus be considered as a *pragmatic* reflexive model as it offers only *one* meta level: encapsulators and mailers do not have objects that describe their behaviors. Furthermore, encapsulators and mailers are used only for specific purpose: *communication*. We believe that such a pragmatic model is enough, and the need for a very general reflexive architecture is not obvious at all.

The asymmetric model is too restrictive: The GARF model is intrinsically asymmetric. There is a mailer which represents the server encapsulator at the client node, but there is no mailer at the server node to represent the client encapsulator. The reason of this asymmetry is that the mailer is a generalization of a proxy, and the proxy model is fundamentally asymmetric. For most of the cases we have encountered, the GARF asymmetric model suffices to describe the desired behaviors.

Nevertheless, there is a specific, yet important, case where this asymmetric model is too restrictive. This case is related to replication and the fact that objects are at the same time clients and servers. More precisely, when both a client and a server are actively replicated, there is a risk of useless invocation replication. Each replica of the client would invoke all the server replicas. If the server itself invokes another server, we will have a proliferation of useless invocations. This may even lead to inconsistencies if the invocations perform state changes. This problem is discussed in details in [17, 18], and we will sketch the way we plan to handle it in Section 6.

5.2 Behavioral abstractions

The main abstractions offered by GARF, in the form of encapsulator and mailer classes, have mainly been inspired by the underlying communication toolkit, i.e Isis. These abstractions represent various point-to-point and multicast communication primitives. We first discuss in the following the lack of orthogonality of these abstractions and its impact on programming. Second, we discuss the fact that two fundamental behaviors are difficult, if not impossible, to represent with these abstractions: persistence and transactions. Note that this is not due to the GARF model itself, but to the restricted set of abstractions offered.

Behavioral classes are not independent: Whereas the behavioral classes offered by GARF might appear at first glance independent (see Figure 5), in fact they are not. Several pairs (encapsulator class, mailer class) are actually strongly coupled and this implies that the programmer does not have a complete free choice of his combinations. This is actually not surprising as each behavioral class represents a specific abstraction for distributed programming, and these abstractions are rarely orthogonal. For example, the encapsulator class `ActiveReplica` can only be used with the mailer class `Abcast`. This is a consequence of the fact that the active replication strategy relies on a total order multicast primitive.

Persistence: In order to ensure the persistence of an object, one must store the object state on a stable storage at the end of each session, and to restore this state at the beginning of another session. This is for example necessary, in the distributed diary manager application, for the `index` which provides the correspondences between the users and their diaries. The state of the `index` must not be lost if the application stops running.

With the abstractions currently offered by GARF, no implicit support for persistence is offered. One must explicitly design and implement an encapsulator class which stores and restores, to and from stable storage, the object state.

Transactions: The abstractions offered by GARF, such as the class `Abcast`, are very convenient to ensure the consistency of replicated objects. A mailer of the class `Abcast` guarantees that invocations on a replicated object are performed on all replicas or on none of them, and concurrent invocations are performed in the same order. In the distributed diary manager application for example, this is adequate to manage the replication of the `index`.

A limitation of such abstractions is however that they ensure consistency when atomic activities concern a single replicated object, e.g. updating the `index`. There are however cases where an atomic activity concern several replicated objects. In the distributed diary application, "scheduling a meeting" is not handled correctly. Indeed, a meeting is scheduled if the date is free for all the involved users. Hence, scheduling a meeting implies an *atomic* update to several `diaries`. When each diary is replicated, and represented by a group, a meeting implies an atomic activity that involves several groups (i.e a transaction). Ensuring the atomicity of such activity is impossible with the GARF current abstractions. More accurately, the only way to ensure such atomicity is to create a huge super-group which contains all replica groups. This limitation is due to the Isis underlying reliable distributed toolkit (and group-oriented systems in general), where the notions of groups and consistency are strongly coupled. We will discuss in Section 6.2, how porting GARF on the Phoenix [11, 12] platform, will enable to provide such abstractions.

5.3 Implementation

Interpreter vs compiler: In GARF, redirection of invocations is a central issue, since the clear separation of functional and behavioral aspects relies on this mechanism. The GARF runtime performs these redirections while objects execute. The GARF runtime can thus be seen as an interpreter in charge or redirecting parts of the invocations. A compiler approach, based on a preprocessor, would perform redirections at compile time. The choice of an interpreter approach was motivated by our will to build a first prototype as quickly as possible. Smalltalk was a good choice for this purpose. Its reflective facilities, e.g the `doesNotUnderstand:` mechanism, and the flexibility in debugging, confirm the commonly agreed assertion that Smalltalk is very well suited for prototyping. Of course, basing GARF on the Smalltalk virtual machine has a negative impact on performance. However, we were quite surprised to see that although not optimized at all, GARF gives acceptable response times, as far as user driven applications (e.g diary manager) are concerned.

The underlying toolkit: Basing GARF on an existing reliable distributed toolkit, i.e Isis, presents several advantages. Reliable multicast primitives are easily implemented (at the object level) since they were all provided by Isis (at the Unix process level). The current version of Isis can be considered as a very stable technology, and our prototype has benefited from this stability, respecting the rule: "never use a prototype to implement a prototype".

However, as we have mentioned in the previous section, group-oriented systems like Isis make it difficult to ensure the consistency of activities involving several replicated objects (i.e transactions in the diary manager). Furthermore, the use of an underlying toolkit that manages groups at the Unix process level, poses the object groups/process groups mapping problem which, due to the difference in granularities, hamper performance. As we will discuss in Section 6, the porting of GARF on Phoenix will enable to handle transactions, and also to avoid the object/process mapping, since Phoenix offers groups at an object level.

Finally, as both Isis and Smalltalk have their own package of threads, we have decided to run both systems inside two different Unix processes (in order to avoid inter-schedulers interferences). This avoids rewriting the Isis scheduler, but the additional communication between the two processes was a performance penalty.

6 Perspectives

In this section, we discuss some perspectives on further design and implementation of GARF.

6.1 Towards a symmetric model

We intend to extend the GARF model, in order to deal with situations such as replication of clients and servers. The basic idea is to come up with a symmetric model where there is a mailer that represents the server at the client node, and a mailer that represents the client at the server node. This model will be used to express the pre-filtering behavior [17, 18]. The pre-filtering consists in avoiding a useless replication of invocations when a client and its server are both replicated. The client mailers will actually be used to filter the invocations, in such a way that only one invocation is forwarded to the server's encapsulator.

6.2 Porting GARF on Phoenix

Extending the basic abstractions: The major characteristic of the Phoenix platform [11, 12] is that it decouples the notion of group from the consistency issues. Phoenix provides the notion of groups to represent replicated data for example, but groups are only viewed as an addressing capability. Reliable communications with consistency guarantees are designed in an orthogonal way to the notion of group. One can thus perform an atomic multicast on a set of group, without creating a super-group for this purpose. This enables to support transactions accessing several replicated objects, such as scheduling a meeting in the distributed diary application.

Direct mapping of object groups: Phoenix is an object-oriented system in two senses [4]. First, it is itself implemented in an object-oriented language and it has a modular and extensible structure. Second, it is aimed to support an object-oriented environment. More precisely, Phoenix offers communication and structuring mechanisms at an object level (rather than at a Unix process level as in Isis). Hence, objects groups in GARF will be directly mapped on Phoenix groups which will avoid the current performance penalty due to the object groups/process groups mapping.

6.3 A pre-compiler approach

As we have mentioned, the interpretor approach has permitted us to implement in a relatively short time a first prototype of GARF. This approach leads however to some performance limitations. For example, the 70 factor overhead is due to the dynamic redirection of invocations. Since a dynamic association scheme between data objects and behavioral objects is not actually necessary, we will experiment a pre-compiler approach. The basic idea is to use a pre-compiler in order to perform the association between data objects and behavioral objects at compile time, Hence no redirection will be necessary during the execution.

Among other optimizations, we will also integrate the Smalltalk system and the Phoenix system within the same Unix process. This will avoid extra communications between Unix processes.

7 Concluding remarks

The main motivation in constructing GARF was to simplify the design and implementation of reliable distributed applications. We wanted to enable such design and implementation at a higher level than existing reliable distributed toolkits such as Isis [2]. We believe that this goal is reached. The distributed diary application was initially written in our Lab, directly in Isis (before GARF was designed and implemented), and we could appreciate the difference in writting the application with GARF.

GARF enforces an incremental programming methodology, where a reliable distributed application can be designed and implemented in several steps and by different programmers. The GARF architecture enhances software modularity, since different aspects of an application are implemented within independent classes: data classes implement functional features, encapsulator and mailer classes implement behavioral features, and system classes implement low level distribution and reliability mechanisms. This enables a programmer to modify the implementation of a behavioral class, e.g. a multicast class, without affecting the rest of the application. GARF also provides flexibility since new behavioral abstractions can be defined and added to the library. For example, by combining the mailer classes *Arpc* (asynchronous rpc) and *Mcast* (multicast), we can define a new mailer class that performs asynchronous multicasts. One of the main lesson learned from using GARF is that a pragmatic reflexive model is enough for a wide area of applications. In GARF, reflection is applied only to inter-object communication, and there is no recursivity in the model, i.e no meta meta object.

The main limitation of GARF is its abstraction library which we believe is too restrictive. As we have said in the paper, this library do not support, for example, a transaction accessing replicated objects. The porting of GARF on the Phoenix [11, 12] platform, which provides a broad range of communication primitives, will enable us to extend the GARF library and support various transaction models. Another limitation is the asymmetry of the programming model. We will extend this model towards a symmetric scheme, and experiment the new model for replication situations. Finally, the current prototype suffer from several drawbacks that hamper performance. We have sketched some possibilities to eliminate these drawbacks.

We are currently designing a new application with GARF: *a virtual private network*. This application is intended to allow the building of a logical private network by using the physical public network infrastructure instead of dedicated network resources (i.e leased lines).

Acknowledgements

A. Schiper, F. Pacull, B. Muganga and R. Mattman have all contributed in the GARF project. The GARF system is currently used in practical teaching of distributed computing (Msc projects) and we have greatly benefited from the feedbacks of the students.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. *Transis: A Communication Sub-System for High Availability.* In Proceedings of International Symposium on Fault-Tolerant Computing. July 1992.
2. K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press. 1993.
3. F. Cristian. *Understanding Fault-Tolerant Distributed Systems.* In Communications of the ACM. February 1991.
4. P. Felber and R. Guerraoui. *Group Programming: an Object-Oriented approach.* In Proceedings of International Conference on Technologies of Object Oriented Languages and Systems. Prentice Hall. March 1995.
5. B. Garbinato, R. Guerraoui, and K. Mazouni. *Programming Fault-Tolerant Applications Using Two Orthogonal Object Levels.* In Proceedings of International Symposium on Computer and Information Science. November 1993.
6. B. Garbinato, X. Défago, R. Guerraoui, and K. Mazouni. *Abstractions pour la programmation concurrente dans GARF.* In “Calculateurs Parallèles” Journal, Hermes publisher. June 1994.
7. B. Garbinato, R. Guerraoui, and K. Mazouni. *Distributed Programming In GARF.* In Object-Based Distributed Programming. Springer Verlag, LNCS 791. June 1994.
8. B. Garbinato, R. Guerraoui, and K. Mazouni. *Implementation of the GARF replicated objects plateform.* In Distributed Systems Engineeting Journal, (2). 1995.
9. A.J Goldberg and A.D Robson. *SMALLTALK-80: The Language and its Implementation.* Addison Wesley publisher. 1983.
10. R. Guerraoui, B. Garbinato, and K. Mazouni. *The GARF Library of DSM Consistency Models.* Proceedings of ACM SIGOPS European Workshop on Operating Systems. September 1994.
11. R. Guerraoui and A. Schiper. *Transactional model vs Virtual Synchrony model: bridging the gap.* In Theory and Practice in Distributed Systems. Springer Verlag, LNCS 938. 1995.
12. R. Guerraoui and A. Schiper. *A Generic Primitive to Support Transactions on Replicated Objects in Distributed Systems.* In Proceedings of IEEE Future Trends on Distributed Computing Systems. Korea, 1995.
13. G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Metaobject protocol.* The Mit Press, 1991.
14. M. Little, D.L. McCue, and S.K. Shrivastava. *Maintaining Information about Persistent Replicated Objects in a Distributed System.* In Proceedings of IEEE International Conference on Distributed Computing Systems. May 1993.
15. S. Maffeis. *Making Distributed Programs Object-Oriented.* In Proceedings of Usenix Symposium on Experiences with Distributed and Multiprocessor Systems. September 1993.
16. C. Malloth and A. Schiper. *View Synchronous Communications on Internet.* In Proceedings of 2nd Open Workshop of the Esprit Project BROADCAST (6360), 1995.
17. K. Mazouni, B. Garbinato, and R. Guerraoui. *Building Reliable Client-Server Software Using Actively Replicated Objects.* In Proceedings of International Conference on Technologies of Object Oriented Languages and Systems. Prentice Hall. March 1995.

18. K. Mazouni, B. Garbinato, and R. Guerraoui. *Filtering Duplicated Invocations Using Symmetric Proxies.* In Proceedings of IEEE International Workshop on Object Orientation in Operating Systems. Lund, Sweden, August 1995.
19. S. Mishra, L. Peterson, and R. Schlichting. *Implementing Fault-Tolerant Replicated Objects Using Psync.* In Proceedings of IEEE Symposium on Reliable Distributed Systems. October 1989.
20. S. Shrivastava. *Lessons Learned from Building and Using the Arjuna Distributed Programming System.* In Theory and Practice in Distributed Systems. Springer Verlag, LNCS 938. 1995.
21. Y. Yokote. *The Apertos reflective Operating System: The Concept and its Implementation.* Proceedings of ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications. 1992.
22. M. Wood. *Replicated RPC Using Amoeba Closed Group Communication.* Proceedings of IEEE International Conference on Distributed Computing Systems. May 1993.

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