Implementation of the GARF* Replicated Objects Platform

Benoît Garbinato Rachid Guerraoui Karim R. Mazouni

Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
Lausanne, Suisse
e-mail: garf@lse.epfl.ch

Abstract

This paper presents the design and implementation of the GARF system, an object-oriented platform that helps programming fault-tolerant distributed applications in a modular way. The originality of GARF is to separate a distributed object into several objects, the complexity of distribution and fault-tolerance being encapsulated in reusable classes. The use of those classes by the GARF system is based on a run-time mechanism of invocation redirection, where most other systems use inheritance, a compile-time mechanism. Our runtime, which supports the GARF object model, is written in Smalltalk. It is presented in detail, as well as the reusable classes that support fault-tolerance. Fault-tolerant objects are implemented using groups of replicated objects. Our *Dependable Object Toolkit* provides group management facilities at the object level. Object groups are built on top of the Isis toolkit, which provides group management facilities at the Unix process level. Our mapping of object groups on process groups and our interfacing of Smalltalk and Isis are detailed. Performance analysis and a first evaluation of our prototype are also presented.

1 Introduction

Programming fault-tolerant distributed applications is a difficult task, because one has to deal with complex issues, such as failure detection, replication management and group communications[1]. GARF is an object-oriented environment that simplifies the programming of fault-tolerant applications, by separating the distributed behaviors of objects from their functionalities [12]. The *functionality* of an object is defined by the part of its code programmed to work in a centralized sequential context. The *behavior* of an object is defined by the part of its code that deals with issues to consider when the object is used in a distributed context. Fault-tolerance is such an issue.

1.1 Overview of fault-tolerance in GARF

The originality of the GARF approach is to separate each distributed object into two independent objects: a *data object*, in charge of the functional aspects, and a *behavioral object*, in charge of the

*Research funded by the Fonds National Suisse pour la Recherche Scientifique, under contract number 5003-034344.

[1] Replication of critical software components, combined with group communications are powerful tools to achieve fault-tolerance, using no specialized hardware [7].

behavioral ones. By doing this, GARF achieves the separation of ideal programming abstractions and efficient implementation; this separation is sometimes called *separation of concerns* [34]. This approach implies that there are two distinct programming levels: programming at the *functional* level and programming at the *behavioral* level. It has the advantage to greatly improve modularity and reusability, since all the difficult aspects are dealt with, once and for all, in the behavioral object classes. With the GARF environment, one first programs and tests data object classes in a conventional object-oriented language. Then, for each data object class, a behavioral object class is chosen from a library and associated to it. The data objects can then be used in a distributed context and have some well-defined behavior, which depends on its associated behavioral object. When developing applications with GARF, one only programs the data object classes, without bothering about any distributed issues. GARF offers a library of ready-to-use components, the behavioral objects, in charge of those issues. The behavior library offers a variety of classes, providing adequate support for fault-tolerance through replication, as well as support for other aspects of distributed programming, such as concurrency [11]. Our model, based on two object levels, also comes with a programming methodology [22].

Behavioral classes that support fault-tolerance at the object level are implemented using the *Isis toolkit* [3], which provides fault-tolerance at the *Unix process* level. The GARF environment uses Isis services through a well-defined interface which is independent of Isis. This approach has the advantage to avoid rewriting algorithms for group management and group communications. It also allows to change the underlying group model very easily, replacing Isis with some other similar toolkit, such as *Horus* [28] or *Transis* [1].

1.2 Overview of this paper

This paper presents the architecture of the current GARF environment prototype and focuses on how fault-tolerant objects are implemented using our two object levels. It also shows how we implemented replicated objects on top of the Isis toolkit, in order to achieve fault-tolerance. Section 2 reviews the GARF computational model, i.e., how data objects and behavioral objects are created, bound together and how they cooperate. Section 2 ends with an overview of the current prototype's architecture. Section 3 presents how the computational model is implemented by the runtime of our prototype. Section 4 presents our underlying dependable object toolkit, its architecture based on Isis, and how it is used to implement fault-tolerant behavioral objects. Section 5 discusses the performance of the GARF environment. Section 6 evaluates the main design choices that prevailed in the implementation of our first prototype of the GARF system. Section 7 describes related work in the distributed object community. Section 8 summarizes what has been done in our prototype and presents the future extensions we plan for the GARF environment.

2 Computational model and basic architecture

The goals of this Section are twofold. In Section 2.1, we present the computational model of GARF and its two object levels. Object invocations and object creations are explained in detail and the main classes and methods needed when programming with GARF are presented. Section 2.2 introduces the architecture of our first implementation of the GARF system and gives an overview of how this architecture supports fault-tolerance.

2.1 Computational model

With GARF, a distributed object is built from two distinct objects: a *data object*, which defines its functionality, and a *behavioral object*, which defines its behavior; the behavior of an object deals with all the distribution related issues. Once a data object class is implemented and tested in a centralized sequential environment, its code will remain unchanged for the rest of the development process. Distributing the instances of some data object class is achieved merely by associating a class of behavioral objects, taken from a library of reusable classesw, to the data object class, that depend on the application semantics. As a result, whenever a data object is created, a behavioral object is automatically created and bound to it.

A behavioral object intercepts all invocations sent or received by its associated data object *transparently to its data object.* Figure 1 shows the client/server invocation scheme[2] as it is perceived by the data objects (dashed arrow) and the invocation scheme as it is really executed through behavioral objects (solid arrows). Each arrow represents an *invocation*, i.e., a (request,reply) pair.

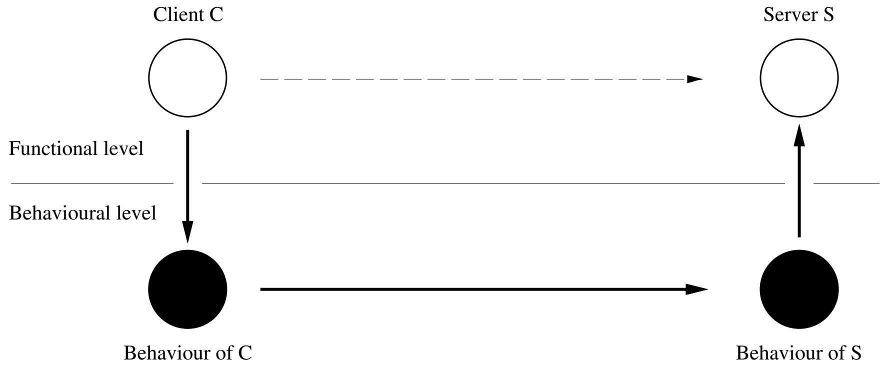

Figure 1: Functional and behavioral object levels

A behavioral object is only a concept and has no real existence. This concept embraces two kinds of real objects: *encapsulators* and *mailers*. In this paper we sometimes use the term *behavioral object* or simply *behavior* to refer to encapsulators and mailers indiscriminately. Each data object has a dedicated encapsulator and both are located on the same site. A copy of the data object's mailer resides on every site where at least one *potential client* is located. An object is a potential client of a distributed object when it has the means to invoke it. There are only two ways for a client to get the mailer of a server: either by creating the server or by receiving its mailer as argument from another client.

Object invocations

When a distributed object participates in an invocation as *client*, only its encapsulator is involved; when a distributed object participates in an invocation as *server*, both its mailer and its encapsulator are involved. Only encapsulators can be invoked across the network and the corresponding mailers alone can do it. The mailer is cloned for each invocation, i.e., a mailer is only responsible for one invocation.

Figure 2 presents the invocation path in the GARF computational model. An invocation of some method **m** on server s by client c is transformed by the GARF runtime into an invocation of method **outRequest:to:** on *En(c)*, the client's encapsulator; the latter receives the following arguments: the reified method #m and *Ma(s)*, a clone of the mailer of server s[3]. The clone is dedicated to that particular invocation. Encapsulator *En(c)* transmits the reified invocation #m to *Ma(s)*, by calling **sendRequest:**. The mailer then calls **inRequest:** on *En(s)*, the server's encapsulator, in order to forward #m to the server's site; this can be seen as some kind of *remote procedure call*. Eventually, *En(s)* invokes **m** on server s. Once method **m** has been performed

[2] Object-oriented languages support the client/server model in a natural way. We often refer to objects as *clients* or *servers*, depending whether they sent or received a particular invocation.

[3] In this paper, we note objects in regular typeface and methods in bold typeface. In Smalltalk syntax (see Figure 2), *En(c)* **outRequest:** #m **to:** *M(s)* means that object *En(c)* is invoked by method **outRequest:to:** with objects #m and *M(s)* as arguments. To *reify an invocation* means to *transform it into an object*; a reified invocation is noted #m, **m** being the method name.

by s, the result comes back to c following the reverse path, i.e., it comes back through a chain of terminating invocations. All actions that are undertaken within **sendRequest:** and **inRequest:**, before forwarding **m** and after receiving its result, define the behavior of s as server. All what **outRequest:to:** does defines the behavior of c as client.

Object creations

Associating a behavior to a data object comes down to create a mailer and an encapsulator, and to bind them to the data object. This tells the GARF environment that henceforth all invocations sent or received by the data object should be reified and forwarded through the invocation path presented in Figure 2.

The association takes place at creation time. When a data object class is invoked for creation, using some creation method such as **new**[4], the invocation is transformed by the GARF system into a call to method **garfNew:**, passing it the reified invocation #new. Method **garfNew:** is known only at the behavioral level. Any implementation of **garfNew:** must create a data object, an encapsulator and a mailer, bind them together and return the mailer. The choice of the encapsulator and the mailer classes determines the behavior of all instances of the data object class. By overriding this method, programmers can specify, for a given class, what the behavior of its instances will be.

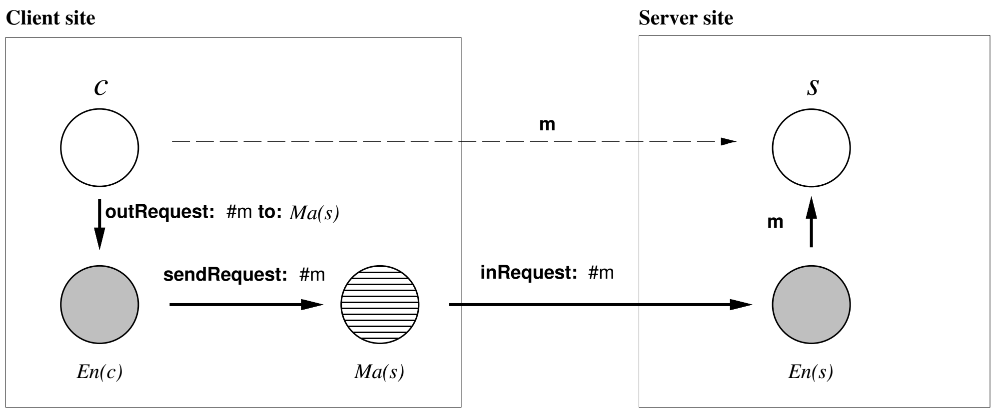

Figure 2: Invocation scheme

Transparency issues

At the functional level, a client data object is not aware that it is invoking a server data object *located on a remote site*: this issue is entirely dealt with at the behavioral level. The remote invocation at the functional level is then transparent. At the behavioral level, only the server's mailer knows where the server's encapsulator is located: the location of the server's encapsulator is transparent to the client's encapsulator. The mailer can be seen as an *intelligent proxy* [2] of the server's encapsulator for the client's encapsulator.

Methods of data objects, encapsulators and mailers

The GARF computational model relies on a set of classes and methods which are presented in Figure 3. Each class is the root of a distinct hierarchy and each method can be redefined by the subclasses. Programming with GARF typically involves subclassing DataObject into several application related classes and reusing encapsulator and mailer classes, derived from Encapsulator and Mailer respectively. These classes are organized into two separate inheritance hierarchies

[4] In the object-oriented language considered here, classes are *first class objects*, i.e., they can be invoked like any object. Object creations are then achieved merely by invoking classes using adequate (creation) methods.

that make up the GARF library of behaviors. Implementation of methods **outRequest:to:** and **inRequest:** for a encapsulator and **sendRequest:** for a mailer determines the distributed behavior they support. Implementation of method **garfNew:** for a data object class determines what behavior, i.e., what encapsulator and mailer, will be bound to its instances.

CLASS	INVOCATION METHODS	CREATION METHODS
DataObject		**garfNew:** aCreationInvocation
Encapsulator	**inRequest:** anInvocation **outRequest:** anInvocation **to:** aMailer	
Mailer	**sendRequest:** anInvocation	

Figure 3: Methods of data objects, encapsulators and mailers

2.2 Basic architecture for fault-tolerance

Our current prototype of GARF is an extension of the Smalltalk [13] *environment*[5] based on a set of additional classes we wrote; no extension to the Smalltalk *language* was necessary to support our computational model. Some of thoses additional classes use operating system level services and the *Isis toolkit* [3] in order to provide support for distribution and fault-tolerance at the object level. Figure 4 shows the basic architecture of our current prototype, with respect to fault-tolerance. Each software layer depends on the layers below to provide fault-tolerance to the layer above. The three first layers implement the computational model presented previously. Next Section details layer 2, while Section 4 details layer 4 and how it is used by layer 3.

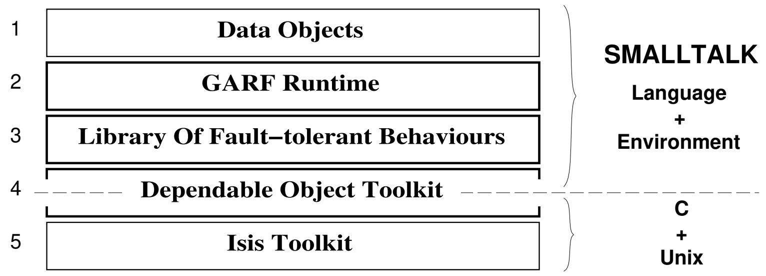

Figure 4: Architecture of the GARF environment

Fault-tolerant data objects rely on the GARF runtime, first to create their behavioral objects and second to redirect all incoming and outgoing invocations. The GARF runtime knows nothing about distribution or fault-tolerance. It merely redirects invocations and relies on the behavioral object library to provide fault-tolerant behaviors. Fault-tolerant behavioral objects are implemented on top of the *Dependable Object Toolkit* (*DOT*). DOT is a platform, developed within the GARF project, that provides support for replication management and group communications at the object level. It is built on top of the Isis toolkit, which provides the same facilities at the Unix process level.

3 The GARF runtime

The GARF runtime implements the computational model presented in Section 2 by carrying out two tasks. First, whenever a data object is created, the GARF runtime creates in addition an encapsulator, a mailer and binds them to it. This comes down to transform any creation

[5]The environment used for the GARF project is VisualWorks, the commercial version by ParcPlace Systems, Inc.

invocation on a data object class into a call to the **garfNew:** method on that class. Second, any invocation sent to some data object s by some data object c is transformed into a call to the **outRequest:to:** method of *En(c)*, the client's encapsulator. Being very similar, both tasks are implemented the same way: by redirecting invocations. The current GARF prototype is based on a *runtime* invocation redirection mechanism, rather than on a preprocessor. This implies that code at the functional level isn't modified at all to implement the redirection.

3.1 Invocation redirection

In order to redirect invocations at runtime, those are *intercepted* by the GARF runtime. This is done by substituting a special object for each data object and data object class; we call this special object a *name*. A *name* is neither a data object nor a behavioral object, and can be seen as a *system object*, since it uses *system services*[6] to help implement the GARF runtime. When client c invokes server s, it really invokes the object *Name(s)*. The latter intercepts the invocation, instead of reacting to it.

The interception is based on an exception mechanism provided by Smalltalk. In this language, whenever an object is invoked using some unknown operation, method **doesNotUnderstand:** is called instead and the invocation that raised the exception is reified and passed to it. Default implementation of this method is inherited from class Object and automatically launches a source level debugger. By making names able to react only to method **doesNotUnderstand:** and by redefining it, invocations can be redirected. Figure 5 presents how the exception mechanism is used in GARF. In this Figure, server s can be either a data object or a data object class.

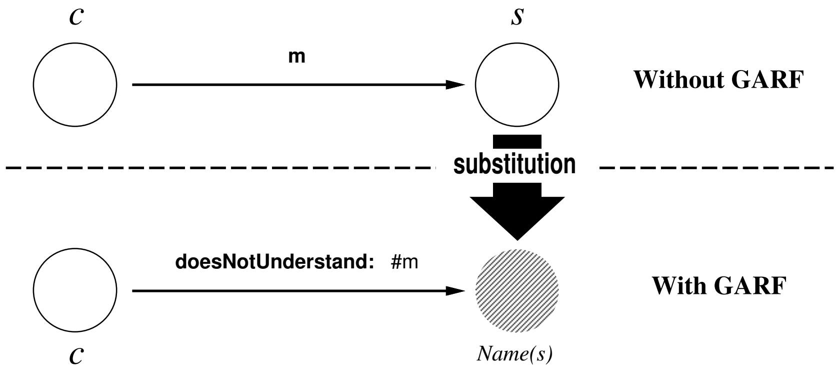

Figure 5: Invocation interception at runtime

Name is an abstract class defined by GARF that provides support for invocation interception. It has no superclass, i.e., its superclass link is set to nil, which ensures that no method is inherited from class Object[7]. Instances of Name's subclasses can then intercept invocations of all methods, apart from the only one they define: **doesNotUnderstand:**. ObjectName and ClassName, described below, are such subclasses. An ObjectName substitutes for a data object in order to intercept its incoming invocations, while a ClassName substitutes for a data object *class* in order to intercept object creations. They both inherit of instance variable namedObject from class Name. Instances of ClassName have their internal variable namedObject that references a class, while in instances of ObjectName, variable namedObject references a mailer.

The interception technique presented above has been used by the Smalltalk community to modify the syntax of the language, in order to extend it towards multiple inheritance [17]. It was also used

[6] By *system services*, we mean the Smalltalk reflective facilities as well as operating system level services. Isis provides such system services and is used to implement system objects supporting fault-tolerance, as we shall see in Section 4.

[7] In Smalltalk, the only class to have a nil superclass is normally Object.

to introduce some semantics changes [5, 10, 26]. There are several problems to solve when a class has nil as superclass. These problems have to do with how the Smalltalk environment (browser, debugger, etc.) interacts with such a class [23].

3.2 Class ClassName

Creation invocations are transformed into calls to method **garfNew:** which is responsible for associating a behavior to each data object when it is created. ClassName instances are responsible for that. They substitute in the SmalltalkDictionary[8] for the classes whose invocations they redirect. The Smalltalk environment always uses the SmalltalkDictionary to access classes, so whenever an object invokes a class, its ClassName is invoked instead. Since the latter is a subclass of Name, it *does not understand* the invocation and its implementation of method **doesNotUnderstand:** is called.

```
doesNotUnderstand: anInvocation
    | result |

    (anInvocation isCreation)
        ifTrue: [result := namedObject garfNew: anInvocation]
        ifFalse: [result := namedObject perform: anInvocation].

    ^result
```

Figure 6: Method **doesNotUnderstand:** defined by ClassName

Figure 6 presents the implementation of **doesNotUnderstand:** by ClassName. If an instance of ClassName receives an invocation that is a creation, **garfNew:** is invoked on the data object class (instance variable namedObject); else the intercepted invocation is merely forwarded. Method **perform:** takes a reified invocation as argument and performs it on the receiver object, i.e., namedObject **perform:** #m is equivalent to namedObject **m**. Figure 7 (a) illustrates the redirection of a creation, while Figure 7 (b) illustrates the redirection of an invocation that is not a creation.

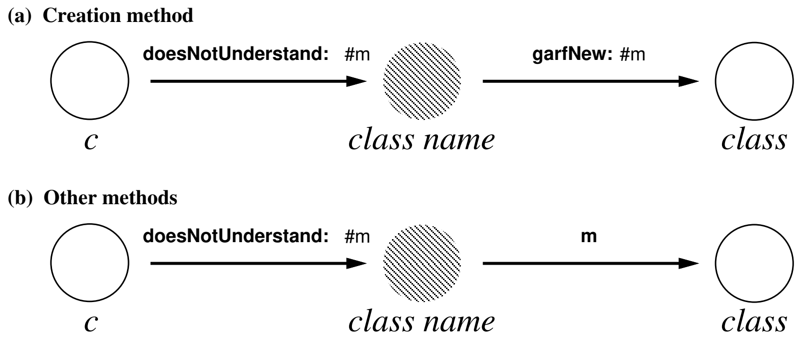

Figure 7: Invocation interception for classes

3.3 Class ObjectName

Invocations of a data object by some client are transformed into invocations of the client's encapsulator by method **outRequest:to:**, which starts the invocation scheme presented in Figure 2.

[8] The SmalltalkDictionary is a predefined global variable containing the collection of all the classes known in the Smalltalk environment. The substitution is currently done by hand, but programming utilities that will ease this task are under development.

ObjectName instances are responsible for that. The ObjectName of a new data object o is created together with o, whose invocations it redirects. Method **garfNew:** returns o's ObjectName to the client instead of directly returning o. Since o might be located on a different site, its ObjectName can be seen as its proxy, just as a mailer was said to be the proxy of a remote encapsulator (see Section 2).

```
doesNotUnderstand: anInvocation
    | client clientEncaps serverMailer |

    client := self getClientOf: thisContext.
    clientEncaps := client encaps.
    serverMailer := namedObject clone.
    ^clientEncaps outRequest: anInvocation to: serverMailer.
```

Figure 8: Method **doesNotUnderstand:** defined by ObjectName

Figure 8 presents the implementation of **doesNotUnderstand:** by ObjectName. Smalltalk provides no predefined method for an object to know who invoked it. Method **getClientOf:** defined by ObjectName is responsible for that. It receives the current stack frame (pseudo-variable thisContext) as argument and follows the dynamic link until it stumbles on an object's frame[9]; the latter is the client's frame, from which it is easy to extract the client itself. The client, as data object, returns its encapsulator when invoked by method **encaps**. When invoked by method **clone**, the server's mailer (instance variable namedObject) returns a copy of itself. Once the client's encapsulator and a clone of the server's mailer have been obtained, the invocation of method **outRequest:to** is possible. Figure 9 presents the *real* path followed by the intercepted invocation #m, while Figure 1 (a) and Figure 2 illustrate *conceptual* paths.

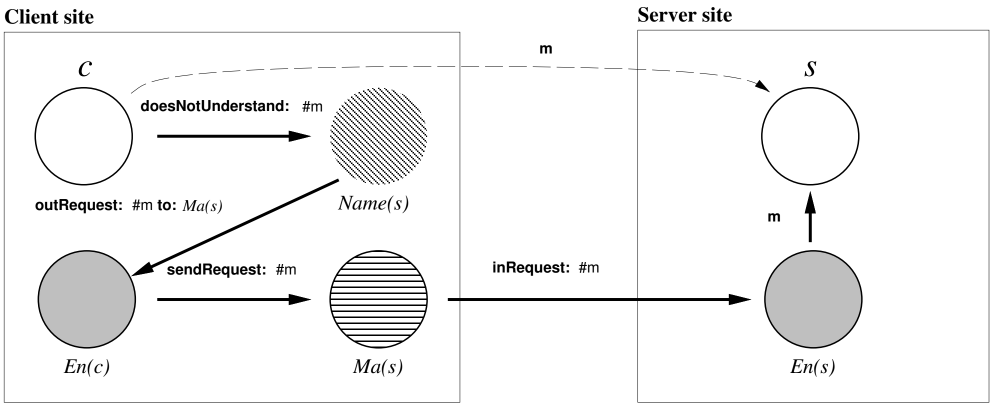

Figure 9: Invocation interception for data objects

4 Support for fault-tolerance

Replication of critical software components has proven to be an efficient way to mask *crash failures* in order to achieve fault-tolerance: a software component is more likely to tolerate failures when replicated. The replicas of a software component are usually managed using *group communication*,

[9] There might be an arbitrary number of nested blocks' frames on the stack before an object's frame is reached, but it is possible to test whether a stack frame has been created by a method call or by a nested block. This implementation is possible thanks to the reflective facilities provided by Smalltalk, such as the ability to manipulate the execution context as an object.

implemented through *multicast primitives* [7]. The concept of group is a powerful abstraction to manage replicas of some logical component, in order to keep it available despite failures (*liveness property*). A useful feature of a multicast is *reliability*: a *reliable multicast* is received either by all non-faulty members of the group or by none [32]. Multicast primitives must ensure that component failures do not compromise the consistency of the logical state managed by group members (*safety property*). By guaranteeing *safety* and *liveness*, an application can be made fault-tolerant, i.e., able to make *safe progress* even if (some) components fail [29]. In GARF, software components are data objects; they are not programmed to deal with replication related issues. Neither does the GARF runtime, which only knows how to redirect invocations. Support for managing groups of objects, and for communicating transparently with their members, is provided by adequate mailer and encapsulator classes, from the GARF library. In this paper, we focus on *active replication*, but several other replication policies are available from the GARF library. Since the GARF library of behaviors does not provide encapsulators supporting persistence yet, we do not consider that aspect in this paper. This Section presents the encapsulator and mailer classes needed to actively replicate data objects (Sect. 4.1) and details the underlying replicated object platform (Sect. 4.2 and Sect. 4.3). This corresponds to layer 3 and layer 4 respectively, as presented in Figure 4.

4.1 Active replication in GARF

Active replication is a technique that requires *each replica* to *receive the requests* sent to the group, to *treat them in the same order* and to *reply* [27]. This guarantees that all replicas have the same observable state[10]. With GARF, an *actively replicated* data object has its behavior built from an encapsulator of class ActiveReplica and a mailer of class Abcast.

Invocation of a replicated object

An ActiveReplica encapsulator is located on each site where a replica of the server data object can be found. It holds a reference to the local replica of the server data object and is member of an object group. Data objects are never group members; their encapsulators are. On the client's site, an Abcast mailer acts as a proxy of the group of encapsulators. When the client's encapsulator invokes **sendRequest:** on the mailer, the latter multicasts method **inRequest:** to the group and each ActiveReplica executes it. Method **inRequest:** invokes the local replica of the server data object, using the reified invocation passed as argument (see Table in Figure 3), and returns the result to the mailer. The Abcast mailer gathers all the replies received from the ActiveReplica encapsulators, but only returns the first one to the client's encapsulator. By redefining method **sendRequest:** in subclasses of Abcast, one could implement other criteria to choose which reply to return.

Figure 10 presents two concurrent invocations of a server data object *s*, replicated twice, by two distinct client data objects *c* and *c'*. Mailers of class Abcast ensure that the multicasts they perform are totally ordered[11]. In this example, both replica of *s* first receive the invocation from *c* (black arrows) and then the one from client *c'* (grey arrows).

Creation of a replicated object

A data object class specifies the fault-tolerant behavior of its instances by redefining its **garfNew:** method (Sect. 2.1). Figure 11 shows how this is done for data objects that are actively replicated twice. First, **garfNew:** creates a set (local variable aSetOfSites) containing the names of the sites where to locate the replicas; those sites are server1.epfl.ch and server2.epfl.ch[12]. This set is passed

[10] Only *deterministic* objects are considered here.

[11] The Abcast class takes its name from the Isis *abcast()* primitive which implements a totally ordered multicast to a group of processes.

[12] The names follow the format defined by the *Domain Naming Service* of Internet. In this example, they are wired in the **garfNew:** method, but they could be the result of some load-balancing computation.

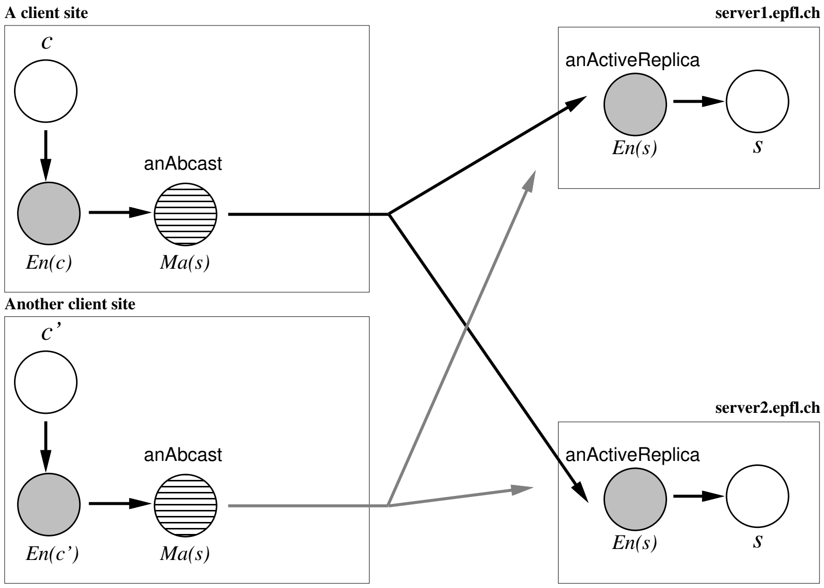

Figure 10: Invocations of an *actively replicated* data object

to method **groupOn:**, which creates and returns an object representing a group of two ActiveReplica encapsulators, one located on server1.epfl.ch and the other on server2.epfl.ch. The group of encapsulators is then invoked using method **buildAndBind:sending:**. This method tells each ActiveReplica to build an instance of the data object class passed as first argument (pseudo-variable self[13]), using the reified creation invocation passed as second argument (parameter aCreationInvocation). Eventually, an Abcast mailer is created, passing it the group of encapsulators (local variable aGroup). This mailer is returned to the caller of **garfNew:** and will act as a proxy of the ActiveReplica group.

```
garfNew: aCreationInvocation
    | aSetOfSites aGroup aMailer |

    aSetOfSites := Set new.
    aSetOfSites add: 'server1.epfl.ch'.
    aSetOfSites add: 'server2.epfl.ch'.

    aGroup := ActiveReplica groupOn: aSetOfSites.
    aGroup buildAndBind: self
           sending: aCreationInvocation.

    aMailer := Abcast to: aGroup.
    ^aMailer
```

Figure 11: Building fault-tolerant behaviors with method **garfNew:**

4.2 Dependable Object Toolkit based on Isis

The Isis toolkit is based on the *virtually synchronous model*. In this model, an application is made of processes (software components) that are members of groups and invoked through multicast

[13] Used within a method, this pseudo-variable represents the object that is currently executing that method, here a data object class executing **garfNew:**. Although named differently, the same pseudo-variable exists in all object-oriented programming languages, e.g., in C++ it is called *this*.

primitives. Members of a group receive a sequence of views, each representing the current group membership. Whenever a process joins or leaves the group, a new view is defined. Messages are ordered with respect to the view changes. By providing tools to manage groups of replicated processes, Isis allows the application to make progress despite process failures (*liveness*). Isis offers three reliable multicast primitives, implementing *total*, *causal* and *fifo* orderings; those primitives are *abcast()*, *cbcast()* and *fbcast()* respectively [3]. The *abcast()* primitive ensures that all processes of a group receive messages in the same order, so that failures do not compromise the consistency of the logical state managed by group members (*safety*). In some cases, a weaker ordering is enough, allowing significant performance improvement.

Developed for the GARF system, the *Dependable Object Toolkit* (*DOT*) is built on top of Isis and offers Isis-like services at the object level. Figure 12 presents the architecture of DOT based on Isis and how it is build using two Unix processes on each site. Details about how those two processes interact are presented in Section 4.3. DOT provides a class Group that allows to create groups of distributed objects and to reliably multicast invocations to them. Members of a distributed object group are replicas of a logical object. The creation method of class Group named **newGroupOf:with:on:** enables to specify a class, a creation method known to that class, and a set of sites where the replicas have to be created. In Figure 11, method **groupOn:** of class ActiveReplica calls **newGroupOf:with:on:** on class Group to create a group of ActiveReplica encapsulators. As a result, instance aGroup representing the group of replicas is returned and can be invoked through multicast methods. Object aGroup can be invoked using one of three multicast methods, **abcast:**, **cbcast:** and **fbcast:**, that have the same ordering semantics as their Isis counterparts. All those methods take a reified invocation as argument which is received and executed by all members of the group. Instance aMailer of class Abcast holds a reference to aGroup and uses method **abcast:** to multicast to it.

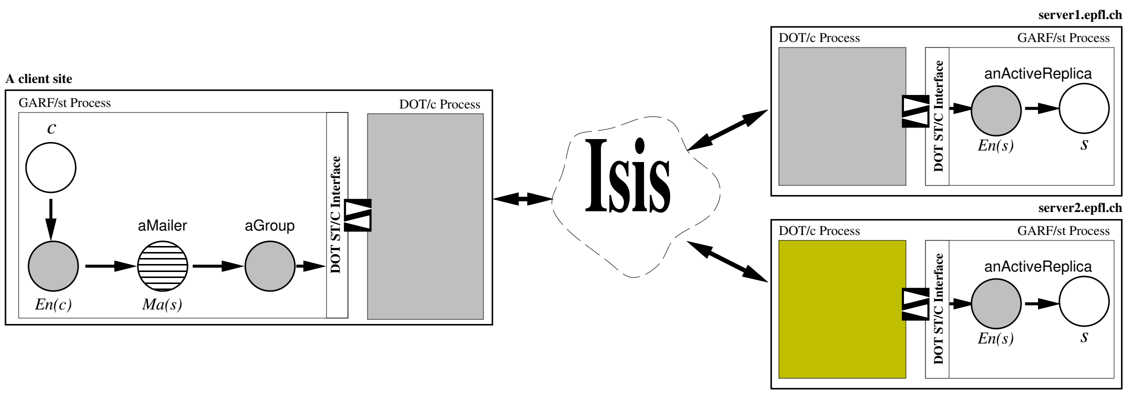

Figure 12: Architecture of DOT based on Isis

4.3 Architecture of DOT

From DOT's point of view, a site is a pair of processes[14]. One process, *GARF/st*, holds the Smalltalk virtual machine where objects execute; Isis knows nothing of this process. The other process, *DOT/c*, is written in C and uses Isis services. Those services rely on a call-back mechanism and on multiple threads controlled by the Isis scheduler. The latter is driven by a protocol that ensures virtual synchrony. Processes GARF/st and DOT/c communicate through Unix domain sockets. Insulating Isis calls in a separate Unix process enhances the modularity of DOT. The latter can be easily ported to another virtually synchronous toolkit, such as Transis or Horus, since

[14] We use the term *process* when talking about *Unix processes* and the term *thread* when talking about *lightweight-processes*.

it only implies rewriting DOT/c. We are currently porting GARF on the Phoenix platform [21], an Isis-like toolkit developed in our laboratory.

Isis manages groups of processes, while DOT deals with groups of objects. Our mapping of DOT groups on Isis groups is straightforward. If a data object replica resides in the GARF/st process of some site and its encapsulator is member of a DOT group g, the DOT/c process on that site is member of an Isis group also named g. So a DOT/c process may be member of several groups: it is member of as many groups as there are data object replicas in the corresponding GARF/st process. Instances of class Group contain the group name g. When invoked by some multicast method (say **abcast:**), they pass g to the DOT/c process through a Unix domain socket. Process DOT/c then uses the equivalent Isis multicast primitive (*abcast()* in that case). The *Group Membership Problem* [30] is entirely dealt with by Isis.

Object group invocations

Figure 13 shows how DOT/c processes interact when an invocation is multicasted to a group of two ActiveReplica encapsulators, using method **abcast:**. As aMailer invokes aGroup, the latter transforms the reified invocation[15] passed as argument into a sequence of bytes. The *Binary Object Streaming Service* (BOSS), provided by the Smalltalk standard library of classes, does the actual job. Object aGroup then sends the sequence of bytes and the group name g to DOT/c, where a dedicated thread reads them (arrow 1) and forks a new concurrent thread (arrow 2). The latter builds an Isis message from the sequence of bytes and calls the *abcast()* primitive, passing it the group name g and the message it just created. Such a call blocks the thread until replies are received back from the replicas on the server's sites.

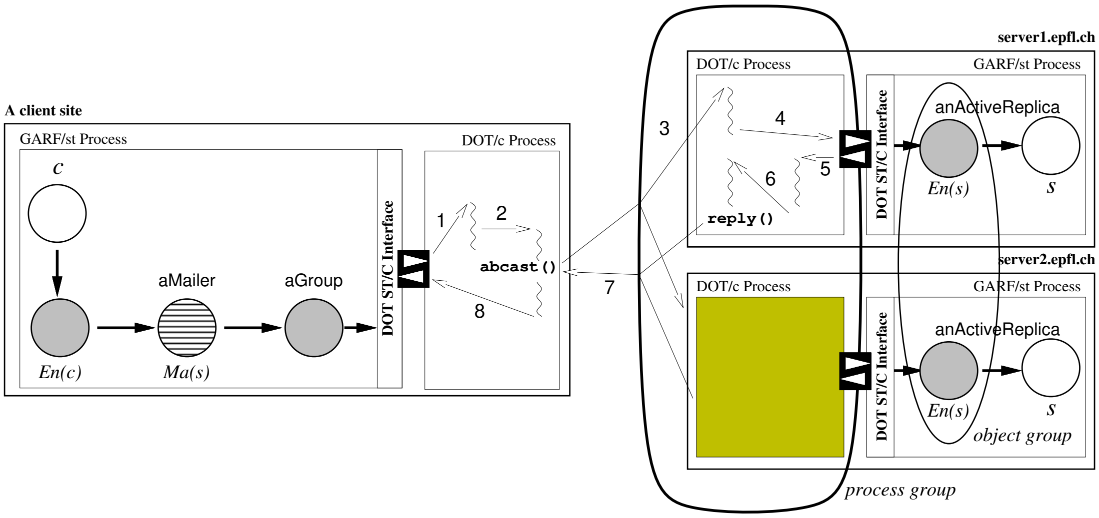

Figure 13: Multicast on a replicated object

As a result of the call to the Isis *abcast()* primitive, a new thread is forked on the server's site[16] (arrow 3). This thread extracts the sequence of bytes representing the reified invocation (**inRequest:**) from the received Isis message. The reified invocation is then the sent to process GARF/st (arrow 4). There, invocation **inRequest:** is rebuilt from the sequence of bytes and performed on encapsulator anActiveReplica. The object resulting from this invocation is transformed into a new sequence of bytes, which is sent back to DOT/c. A dedicated thread reads those bytes (arrow 5)

[15] With GARF, the reified invocation is always a call to **inRequest:** since only groups of encapsulators are used. However, DOT is decoupled from GARF and allows to manage replicas of any class of objects, not only encapsulators.

[16] Only one server site is detailed in Figure 13; the other one does exactly the same (s is actively replicated).

and forks a new concurrent thread (arrow 6). The latter builds an Isis reply and calls the *reply()* primitive to send it back to the client's site (arrow 7). When the replies from all the non-faulty processes have been received on the client's site, Isis unblocks the thread that called the *abcast()* primitive. That thread extracts the sequences of bytes representing the replies and sends them back to process GARF/st (arrow 8), where the reply objects are rebuilt from the sequences of bytes and returned to aGroup. The latter collects all the replies into a list and returns it to aMailer.

Along the path followed by a multicast invocation, many concurrent threads are involved; some are forked to deal with only one invocation. All the threads presented in Figure 13 are Isis threads executing in process DOT/c, but Smalltalk threads are also involved; they execute in process GARF/st and do not appear in the Figure. This approach based on forking multiple threads is useful to avoid serializing concurrent invocations on independent replicated objects.

Object group creations

The *creation* of a group of ActiveReplica encapsulators is very similar to what has been presented for multicast invocations. First, a unique group name is generated, say g. DOT creates an Isis message containing the creation method, the generated group name g and the list of the sites where the replicas are to be located. That message is multicasted to a special group containing all the DOT/c processes. The latter can then determine if they are concerned by the new group creation by looking at the list of sites they received. On each of the sites that are in the list, class ActiveReplica creates a new instance locally, using the received creation method, and the DOT/c process joins a new Isis group named g.

5 Performance measurements

The GARF system current prototype has not been optimized at all yet; it is a straightforward implementation of the design presented in this paper. The performance measurements presented in this Section were carried out using three Sun SPARCstations interconnected through a 10 Mbit Ethernet. All workstations were equipped with 32 Mbytes RAM and were running Solaris 2.2. The measurements took place on normal workday, so the three workstations had medium to high load: all were running XWindows as well as several interactive applications (emacs, mosaic, etc.). The test scenario was based on the example introduced in Figure 13, i.e., some client object interacts with a server object that is actively 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 reply. The client as well as one of the server replica ran on two SPARCstations LX, while the other server replica executed on a SPARCstation 10.

5.1 Performance of the GARF runtime

Before analyzing the performance of the *Dependable Object Toolkit* (*DOT*), lets take a look at the overhead due to the GARF model, compared to a mere Smalltalk invocation. When applying our scenario to "normal" Smalltalk objects (no behavioral objects associated to them, so no distribution and no replication), the response time for a read invocation is 9 μs on a SPARCstation 10 and 26 μs on a SPARCstation LX. Now let's make data objects out of those "normal" objects and bind them to behavioral objects that simply redirect invocations[17]. When intercepted by the GARF runtime and redirected to those behavioral objects, the read invocation takes 628 μs on a SPARCstation 10 and 1.8 ms on a SPARCstation LX; the GARF runtime causes an overhead factor of approximately 70. As we will see in next Section however, that overhead is quite small compared to the response time measured when data objects are distributed and replicated.

[17] Data objects are still *local non-replicated* objects in that case. With GARF, when no specific behavior is defined for some data object class, this simple redirection behavior applies by default to its instances.

The reasons for such a high factor have to do with how the Smalltalk virtual machine executes and how it handles exceptions. During normal execution, the virtual machine is partially bypassed and object methods directly execute on the physical processor. When an exception is raised, the virtual machine *reinterprets* the method that caused the exception, which makes the execution much slower. This is what happens when method **doesNotUnderstand:** is called. Another slowdown factor is the call to method **getClientOf:** which manipulates the execution stack as a Smalltalk object (Sect. 3.3): it requires the virtual machine to dynamically build object representations of real stack frames. One more slowdown factor is the cloning of mailers, which implies memory allocation and copy for each invocation; a simple optimization could be to manage a pool of pre-allocated mailers.

5.2 Performance of DOT

The time spent in the GARF runtime and in data objects is very small compared to the 251 *ms* a read invocation takes when objects are distributed and replicated (less than 1% of the total response time). In this Section, we concentrate on the performance of DOT: no more mention will be made of the GARF runtime's overhead and of the response time of "normal" Smalltalk objects, those times being negligible. Figure 14 shows the decomposition of the response time for our scenario, i.e., an **abcast:** invocation on a group of two ActiveReplica encapsulators. Each site corresponds to a workstation where processes GARF/st and DOT/c are running. The client is on client.epfl.ch (SPARCstation LX), while the server replicas are located on server1.epfl.ch (SPARCstation 10) and server2.epfl.ch (SPARCstation LX). On the client's site, the response time of a call to the Isis *abcast()* primitive is 138 *ms*. That time includes the treatment by processes DOT/c and GARF/st on the server's sites, as well as the Isis communication. The time spent for DOT on the client's site is then 113 *ms* (251 $ms - 138$ *ms*). On the server's site, that time is 67 *ms* for server1.epfl.ch and 91 *ms* for server2.epfl.ch. Being on a local area network, it is reasonable to admit that Isis messages reach DOT/c processes at the same time on both server's sites. We can then say that 204 *ms* (113 $ms + 91$ *ms*) is the total time taken by DOT to perform method **abcast:** on the group of replicated servers. The reason we use 91 *ms* and not 67 *ms* for our calculation comes from the fact that instances of class Group wait for all replies when invoked by method **abcast:**. The deciding response time is therefore the one of the slower server. As a consequence, 81% of the total response time is used by DOT, while the Isis communication takes the remainder (47 *ms*). Compared to the time taken by the Isis primitive alone, DOT causes an overhead factor of approximately 5.

Several simple optimizations could be done to improve performance. One such optimizations would be to avoid waiting for all the replies at the Isis level, since at the GARF level the Abcast mailers only returns the first object from its list of replies anyway. In that case, the response time would be that of the faster server, not of the slower one. Using Isis *cbcast()* primitives, which is roughly twice as fast as *abcast()* primitives, is sometimes enough to insure safety; this is another simple optimization. Further in-depth performance analysis has to be carried out in order to better understand where optimization efforts should be concentrated.

6 Evaluation of GARF's first implementation

We evaluated the design of our first prototype by implementing a fault-tolerant distributed application using GARF: the *Distributed Diary Manager* (*DDM*). DDM, which runs on a set of workstations interconnected through a local area network, is aimed to manage a diary for each user and to allow him to plan meetings with other users. A user can visualize the meetings he is expected to attend as well as the list of all the other users. Interaction with DDM is done through a graphical user interface (made of windows, menus, buttons, etc.) which was built thanks to the facilities of the Smalltalk programming environment. All the objects that hold DDM related

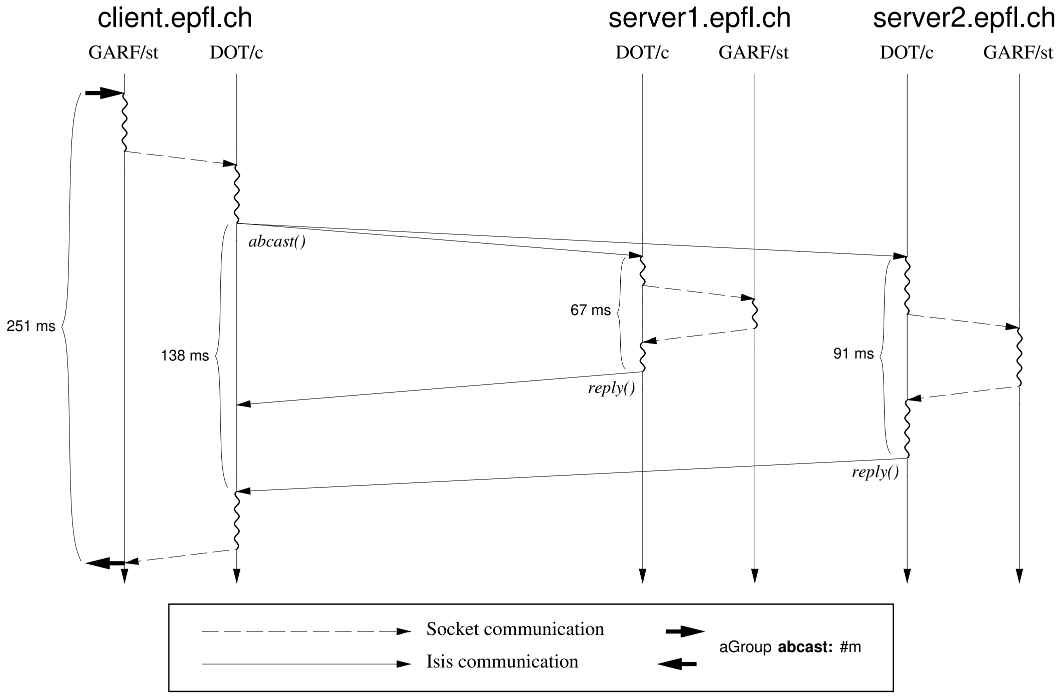

Figure 14: Performance model

data are implemented as data objects to which encapsulators and mailers have been associated. The complete description of DDM's implementation using GARF can be found in [22]. This Section discusses the main design options that lead to our first prototype of the GARF environment, in the light of our experience of programming DDM. Those main options are the choices of Smalltalk (Sect. 6.1) and of Isis (Sect. 6.2).

6.1 The choice of Smalltalk

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 invocation redirections while objects execute and can then be seen as an interpreter in charge of redirecting part of the invocations[18]. A compiler approach, based on a preprocessor, would perform redirections at compile time. The choice of the interpreter approach was motivated by our will to build a first prototype as quickly as possible. Implementing a preprocessor would have implied more code to write and less flexibility when debugging. Since Smalltalk offers a large spectrum of reflective facilities [10], e.g. the **doesNotUndertand** mechanism, the interpreter approach has proven to be efficient in prototyping. It made the coding and debugging of our first implementation very fast, which confirms the commonly agreed assertion that Smalltalk is very well suited for prototyping[19]. 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 (such as DDM) are concerned.

[18] The redirection only concerns invocations involving data objects (subclasses of the DataObject class). As mentionned before though, no extension to the Smalltalk language was necessary to support the redirection; the term *interpreter* is only used here to emphasize the fact that the redirection occurs at run-time.

[19] Development in Smalltalk is known to be faster and to require less lines of code than development in C++ [16].

6.2 The choice of Isis

Building DOT on top of Isis presents several advantages. First, reliable multicast primitives with different ordering semantics are easily implemented, since all the virtually synchronous protocols are managed by Isis. Then, the Isis implementation of the virtually synchronous model has been around for quite a while now and has gone through intensive testing by its users. The current version of Isis can be considered a relatively stable technology and DOT directly benefits from this stability. Finally, both *distribution* and *replication* of objects can be implemented together: a non-replicated distributed object is merely implemented using a group with a single member. DOT's two-processes architecture (Sect. 4.3) allows GARF to take advantage of the optimizations implemented by the Isis toolkit. If DOT had been implemented all in one process, it would have been necessary to rewrite the Isis scheduler, basing it on Smalltalk threads[20], and to rewrite part of the virtually synchronous protocols. Such a design would have implied losing part of the optimizations of the Isis toolkit. It would also have made the development of DOT a great deal more complex!

7 Related work

With the increasing interest in distributed systems and in object-oriented design, many platforms that support programming with *distributed objects* have been developed in recent years. A common approach to achieve separation of concerns is to use inheritance: objects inherit adequate behaviors from a set of predefined classes. *Arjuna* [19], *Avalon* [8] and *Electra* [20] are examples of such systems. Another approach bases the separation of concerns on reflective facilities that rely on two object levels: a *base-level* and a *meta-level*. *Muse* [34], *Open-C++* [6] and of course *GARF* are such systems. In GARF, the base-level and the meta-level correspond to the functional and behavioral levels respectively. A the present time however, few systems provide support for fault-tolerance; only those can be compared with GARF. In this Section, we do not intend to list all object-oriented platforms that provide support for fault-tolerance. We chose four significant systems and compared them with GARF; several other platforms are only mentioned as similar to one of those four systems.

7.1 Arjuna

In *Arjuna*, means are provided to insure strong consistency on replicated persistent C++ objects [19]. Replication mechanisms are hidden by inheritance and are based on a preprocessor and on stub-object libraries. Predefined classes which user classes can inherit to get adequate behavior, such as persistence and replication, are provided. Arjuna is based on the transactional model, while GARF is based on the virtual synchronous model. Since only one object level is available in Arjuna, programmers have to access replication specific mechanisms directly in their code, e.g., to modify the group membership. In GARF, no use of any replication specific mechanism is made within data objects' code. Those aspects are entirely dealt with by the predefined behavioral objects associated through the **garfNew:** method. In that sense, Arjuna does not achieve separation of concerns as GARF does it.

7.2 Electra

Electra as well is an extension of the C++ programming language that offers support for the development of fault-tolerant objects [20]. It is based on the *Horus* toolkit, which relies on the *Multicast Transport Service* (MUTS) [28]. Electra provides very similar abstractions for building fault-tolerant objects to those supported by GARF. Those abstractions are implemented on top

[20] Two different lightweight-process schedulers cannot coexist within the same Unix process.

of multicast primitives to groups of active entities provided by the underlying operating system (lightweight-processes in Electra, Unix processes in GARF). With Electra, a client object need not know that it is invoking a replicated object: the group communication is made transparent by the use of so-called *smart proxies*. Several reusable classes of active objects (called *services*) are provided as well as *thread-safe abstract data types*, that are passive objects used by services. Unlike with GARF, fault-tolerant behaviors are not clearly separated from the rest of the code of an object. If one wants two instances of a same kind of objects (say a set) to have different behaviors (say one is actively replicated twice and the other is passively replicated three times), it is necessary to define two different classes through inheritance. Electra can be seen as a kind of *object-oriented wrapper* for the Horus toolkit.

Several systems provide similar approaches to Electra's. Among these systems, we can mention *ROMANCE* [31] which is also an extension of C++. In this system, a client invokes a local *Group Remote Invocation Proxy* (*GRIP*) which in turns multicasts the invocation to a group of *Ambassadors*. Ambassadors represent the invoking client on the sites where replicas of the server are located. ROMANCE is based on the group technology provided by a platform called *xAMp*. *Psync* [24], *Emerald/Gaggles* [4] and *FOG/C++* [14] are three other systems similar to Electra. FOG/C++ is built on top of the experimental operating system *SOS* [33]. Psync and Emerald/Gaggle have the particularity not to be based on a system level platform to support fault-tolerance through replication. The protocols that guarantee safety and liveness of the applications are directly coded within the object-oriented platform.

7.3 RDO/Smalltalk

Like GARF, the *RDO/Smalltalk* is based on Smalltalk and Isis [18]. RDO/Smalltalk uses an extension of Isis services called *Isis News*, which implements *publish/subscribe* mechanisms for groups of Unix processes. In this model, processes can *subscribe* and/or *publish* to some *subject*[21]; publishing messages to a subject results in multicasting that message to all subscribers. The available ordering criteria correspond to those offered by Isis. In RDO/Smalltalk, objects execute in a Unix process that contains the Smalltalk virtual machine. That process communicates with another Unix process registered as publisher and/or subscriber of some Isis News subject. This two-processes architecture is very similar to the one adopted in GARF. Each group of object replicas corresponds to an Isis News subject. Replicated server objects are accessed transparently through *proxy objects* but created explicitly. Transparent invocations are achieved using the same mechanism as in the GARF runtime (based on method **doesNotUnderstand:**). The proxy class is the main reusable component of RDO/Smalltalk. There is no clear separation of the fault-tolerant behaviors from the rest of the code. The system called *The Information Bus* [25] offers similar publish/subscribe abstractions for non-replicated distributed objects.

7.4 Fault-tolerance with Open-C++

The platform described in [9] is the only object-oriented system that we know of, that clearly separates fault-tolerance in a distinct object level, using reflective facilities. Open-C++ [6] is an extension of the C++ language which adds a *meta-level* to traditional (*base-level*) programming in C++[22]. Objects of the meta-level are called *meta-objects*. They allow the programmer to redefine the way objects interact, much in the same manner the GARF runtime does it. Open-C++ was not primarily designed to support fault-tolerance. However, its reflective meta-level made it very easy to add replication of objects. The implementation described in [9] does not rely on a system level platform to support replication. The protocols that deal with this issue are directly coded

[21] This approach allows to log messages and to communicate with groups that have currently no members.

[22] The C++ language is known to lack reflective facilities. For example, C++ classes are only compile-time entities that no longer exist at runtime. It is therefore impossible to act on how instances of some class send or receive invocations.

within meta-objects. Unlike the GARF runtime, Open-C++ implements a compiler approach, based on a preprocessor.

8 Conclusion

GARF provides high level abstractions to help programmers building object-oriented distributed applications that are fault-tolerant. Its originality is to break each distributed object into two separate objects, one of which manages all the difficult aspects that have to be dealt with when objects are replicated. The latter objects are instantiated from classes of ready-to-use components. Those classes are available from the GARF library of behaviors. Modularity and reusability greatly benefit of this approach. As far as we know, only the GARF system and the fault-tolerant extension of Open-C++ [9] clearly separate fault-tolerance from the other aspects *into two distinct object levels*, but only GARF relies on the Isis platform.

Our first prototype has been implemented in Smalltalk and uses the Isis toolkit to achieve fault-tolerance. It broadly exploits the flexibility of the Smalltalk programming environment to achieve transparency of the behavioral level. Although not optimized at all, our first implementation yields acceptable performance for the *Distributed Diary Manager* application we developed using GARF. Our first prototype of GARF directly benefits from the many optimizations Isis implements in its virtually synchronous protocols. We are now planning to further analyze where optimization could take place. Future work will also consist in studying the advantages of rewriting GARF in C++, in order to move it on the Phoenix platform [21] currently developed in our laboratory. This platform will make it very easy to blend abstractions of the transaction model and of the virtual synchrony model [15].

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, July 1992.

[2] 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, 1994.

[3] K. Birman and T. Joseph. Exploiting Replication in Distributed Systems. In S. Mullender, editor, *Distributed Systems*, chapter 15, pages 319–367. ACM Press, 1989.

[4] A. P. Black and M. P. Immel. Encapsulating Plurality. In *European Conference on Object Oriented Programming Proceedings*. Springer Verlag, July 1993.

[5] J.-P. Briot. Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In *European Conference on Object Oriented Programming Proceedings*, pages 109–129. Cambridge University Press, 1986.

[6] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with Meta-Level Architecture. In *European Conference on Object Oriented Programming Proceedings*. Springer Verlag, July 1993.

[7] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[8] D. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronisation and Recovery Properties in Avalon/C++. *Computer*, 21(12), December 1988.

[9] J.-C. Fabre, V. Nicomette, T. Pérennou, and Z. Wu. Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. Technical Report LAAS-94156, Centre National de la Recherche Scientifique (France), April 1994.

[10] B. Foote and R.E. Johnson. Reflective Facilities in Smalltalk-80. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 327–335. ACM Press, 1989.

[11] B. Garbinato, X. Défago, R. Guerraoui, and K. R. Mazouni. Abstractions pour la Programmation Concurrente dans GARF. *Calculateurs parallèles*, June 1994.

[12] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Distributed Programming in GARF. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object Based Distributed Programming*. Springer Verlag, 1994.

[13] A.J. Goldberg and A.D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison Wesley, 1983.

[14] Y. Gourhant and M. Shapiro. FOG/C++: A Fragmented-Object Generator. In *C++ Conference*. Usenix, April 1990.

[15] R. Guerraoui and A. Schiper. Transaction Model vs. Virtual Synchrony Model: Bridging The Gap. Technical Report 94/62, Ecole Polytechnique Fédérale de Lausanne, August 1994.

[16] P. Haynes. "C++ is Better Than Smalltalk"?? In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proceedings of the 12th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-12)*. Prentice Hall, 1993.

[17] D.H.H. Ingalls and A.H. Borning. Multiple Inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237. AAAI, 1982.

[18] ISIS Distributed Systems Inc. *The RDO Programmers Manual*, August 1993.

[19] M.C. Little and S. K. Shrivastava. Object Replication in Arjuna. Technical report, University of Newcastle, 1994.

[20] S. Maffeis. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming*. Springer Verlag, 1994.

[21] C. Malloth. PHOENIX, A Plateform for Fault Tolerant View Synchronous Communications. In M. Aguilar, editor, *Proceedings of the '94 SIPAR-Workshop on "Parallel and Distributed Computing"*, October 1994.

[22] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Programmation d'une application distribuée résistante aux pannes avec l'environnement GARF. In *Proceedings of the 3rd Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI-3)*, 1994.

[23] P. L. McCullough. Transparent Forwarding: First Steps. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 331–341. ACM Press, 1987.

[24] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing Fault-Tolerant Replicated Objects Using Psync. In *Proceedings of the IEEE 8th Symposium on Reliable Distributed Systems*, October 1989.

[25] B. Oki, M. Pfluegl A. Siegel, and D. Skeen. The Information Bus - An Architecure for Extensible Disributed Systems. *Operating Systems Review*, 27(5), December 1993.

[26] G.A. Pascoe. Encapsulators: A New Software Paradigm in Smalltalk-80. In *Proceedings of the Conference on Systems, Languages and Applications*, pages 341–346. ACM Press, 1986.

[27] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*, volume 1. Springer-Verlag, 1991.

[28] R. Van Renesse, K. Birman, and R. Cooper. The Horus System. Technical report, University of Cornell (NY), 1993.

[29] A. Ricciardi, A. Schiper, and K. Birman. Understanding Partitions and the "No Partition" Assumption. In *Proceedings of the IEEE 4th Workshop on Future Trends of Distributed Systems (FTDCS-93)*, September 1993.

[30] A. M. Ricciardi. *The Asynchonous Membership Problem*. PhD thesis, Cornell University, January 1993.

[31] L. Rodrigues and P. Veríssimo. Replicated Object Management using Group Technology. In *Proceedings of the IEEE 4th Workshop on Future Trends of Distributed Systems (FTDCS-93)*, September 1993.

[32] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually-Synchronous Environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS-13)*. IEEE, May 1993.

[33] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: An Object-Oriented Operating System - Assessment and Perspectives. *Computing Systems*, 2(4), December 1989.

[34] Y. Yokote, G. Kiczales, and J. Lamping. Separation of Concerns and Operating Systems for Highly Heterogeneous Distributed Computing. In *Proceedings of the 6th ACM SIGOPS European Workshop*, September 1994.