GARF: A Tool for Programming Reliable Distributed Applications

Rachid Guerraoui Benoît Garbinato Karim Mazouni
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract

GARF is an object-oriented tool that supports the design and programming of reliable distributed applications. GARF lies at an intermediate level between applications and process group communication systems. It enables a developer to first program an application in a centralized environment, then to distribute it and increase its reliability by replicating its critical components over several machines. Using a built-in library of distributed abstractions, the developer can choose a replication strategy for each component of the application.

Keywords: Distributed programming, replication, group communication, objects.

Programming Reliable Distributed Applications

A well known way to increase the reliability of an application is to replicate its critical components over several machines. While this principle is readily understood, the techniques required to preserve the consistency of a distributed and replicated application pose difficult problems. Group communication systems implement these techniques by providing multicast primitives that deliver messages to various processes (groups of replicas) with atomicity and ordering guarantees. A typical example of such systems is Isis [1].

Although powerful, group communication systems are at a too low level (Unix process level), and are thus very difficult to use for programmers without a broad experience in distributed computing. Furthermore, when using these systems, multicast primitives must be mixed with application *functional* features (i.e. sequential and centralized aspects), hampering extensibility and modularity. As a consequence, an application written in a centralized context cannot be directly distributed and replicated.

This paper introduces GARF: an object-oriented tool that supports the design and programming of reliable distributed applications, at a higher level than existing group communication systems such as Isis. GARF can be viewed as an intermediate between group communication systems and applications, and it is actually implemented on top of Isis. GARF promotes software modularity by clearly separating the *behavioral* features that concern concurrency, distribution and replication, from *functional* features than concern traditional centralized aspects.

GARF encourages an incremental programming methodology. First, the GARF programmer designs and implements 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, distributing the application over a network, and then replicating its critical components. Of course, all the steps, including the first one, may be performed concurrently, but the code written for different steps is separated.

GARF offers an extensible library of abstractions for distributed programming. Among these abstractions are remote object invocations with different semantics, such as *synchronous* and *asynchronous* invocations, and object group invocations with different ordering guarantees, such as *reliable, fifo, causal* and *total order* [9]. These abstractions enable it to support different kind of replication strategies, including *active* replication where all the replicas treat the requests and send back a reply, and *primary-backup* replication where one particular replica (the primary) performs the job whereas the others are only used in the case the primary crashes [2]. An experienced programmer can implement new abstractions by combining existing ones, or by using low level group communication primives underlying GARF.

GARF Programming model

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"* [8], used to describe behavioral features (i.e. concurrency, distribution, and replication) 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. For example, the class `Replica` enables to create multiple replicas of a data object, in order to increase its reliability.
   - **Mailers** are used to perform (remote) communications between encapsulators. GARF provides a library of mailer classes. For example, the mailer class `Abcast` ensures that all replicas of an encapsulator receive concurrent requests in the same order. Using this class, the replicas keep the same state despite concurrent invocations.

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

Figure 1 describes the runtime relation between data objects and behavioral objects. The figure depicts two data objects, `C` and `S`, two encapsulators, `e(C)` and `e(S)`, and a mailer `m(S)`

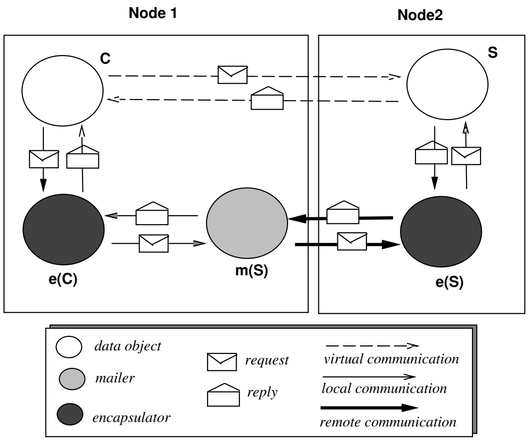

Figure 1: data objects, encapsulators and mailers

which handles the communications between the encapsulators.

The three kinds of objects supported by GARF convey three corresponding programming levels: the *data level*, the *binding level* and 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 concurrency, distribution, or replication.

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

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 classes* [4]. At this level, the programmer must be familiar with the GARF system, and even with the underlying group communication system 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.

Example: a 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 [3]. 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 cancelation 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 whose machine is also up. We do not assume communication link failures here.

Functional level

At the functional level, we designed four main data classes: `DiaryIndex, Diary, Meeting`, and `DiaryService`. Class `DiaryIndex` implements operations returning a diary given a user name, listing the names of all diary owners, and adding or removing a user. This class has one instance: the `diary index`. Class `Diary` implements operations that list all the meetings contained in a diary and update this list. This class has one instance per user. Class `Meeting` implements operations that return a list of participants (in the meeting), return the meeting period and remove a user from the list of participants. 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 for each session, i.e each time a user starts working with the application.

Binding level

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

- **Choosing behavioral classes:** This step depends on the set of application specific characteristics.

  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.

```
          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.
          ...
```

Figure 2: A `garfNew:` operation

5. Since the application is intended to support several users at the same time, the `diary index`, `diaries` and `meetings` may be accessed concurrently.

The encapsulator class `ActiveReplica` provides the desired behavior for replication. It ensures that all replicas treat an invocation, and return a reply. As long as one replica is alive, a reply is returned. No specific treatment is needed if some of the replicas crash. The mailer class `Abcast` ensures that all the replicas are invoked in the same order, and hence ensures that the replicas keep the same state despite concurrent invocations. In other words, the replicas are strongly consistent. Furthermore, a mailer of the class `Abcast` 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. The data object is created on the encapsulator's node.
  3. Creating a mailer and binding the encapsulator to it. The data object (already bound to the encapsulator) is implicitly bound to the new mailer.

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. As shown in the figure, 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.

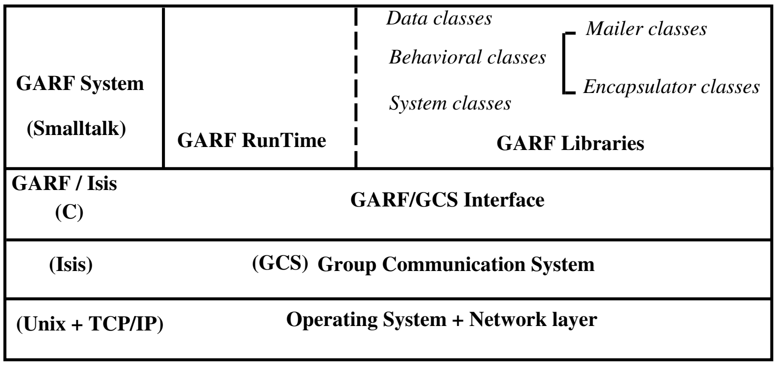

Figure 3: GARF software layers

GARF Architecture

GARF is not intended to be used in a stand-alone fashion, but rather on top of a group communication system providing low level primitives for remote communication, group management, and reliable multicast communication. GARF is written in Smalltalk on top of the Isis group communication system. GARF is composed of a runtime and three class libraries: the *encapsulator* library, the *mailer* library, and the *system* library:

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.

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`.

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).

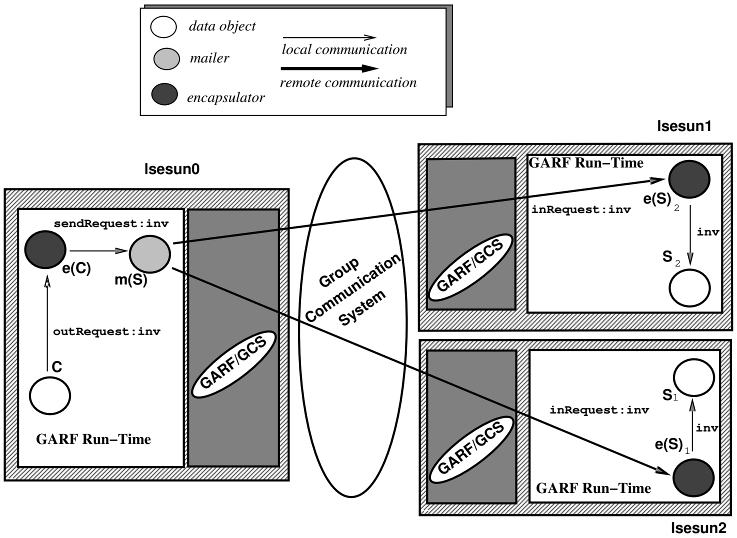

Figure 4: GARF execution model

- 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*.

GARF abstractions

Encapsulator library: encapsulator classes are subclasses of the basic class Encapsulator. This class provides two operations: inRequest and outRequest, which handle the way data objects receive and send requests. Defining encapsulator classes consists in programming these operations inside subclasses of class 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 [4] and [5].

Mailer library: mailer classes are subclasses of the basic class Mailer. This class provides the operation sendRequest, which handles the way communications are performed between encapsulators. Defining mailer classes consists in programming this operation inside subclasses of Mailer. GARF provides a library of mailer classes (Figure 5 (b)), which handle synchronous and asynchronous remote invocations (i.e classes Srpc, and Arpc), as well as various semantics of multicasts (i.e classes Mcast for reliable multicast, Fbcast for *fifo* multicast, Cbcast for causal multicast [7], and Abcast for atomic multicast [7]). These classes are detailed in [4] and [5].

System library: this library contains a set of classes called *system* classes, of which objects represent the main abstractions used to program behavioral classes. System classes also act

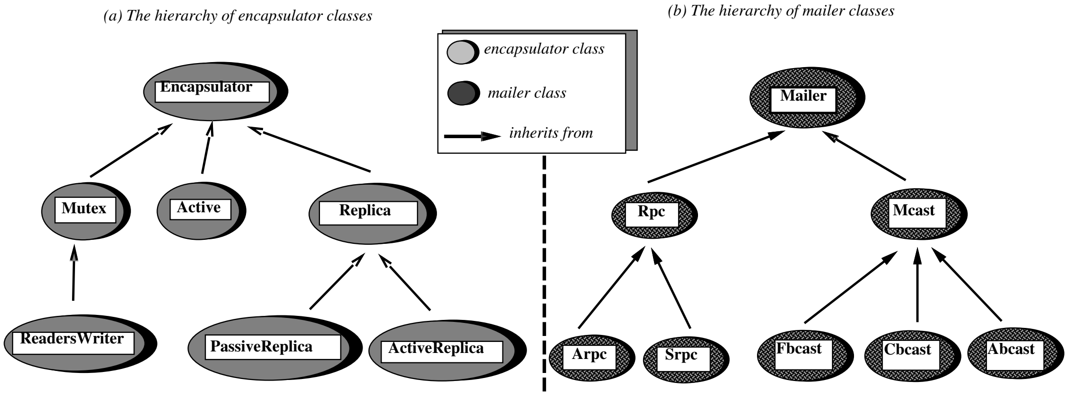

Figure 5: GARF encapsulator and mailer classes

as an interface to the underlying group communication system (actually Isis). System classes provided by GARF are for instance, the class `Group` that handles object group management, the class `Future` for asynchronous invocation, and the classes `ClassName` and `ObjectName` that manage object invocation interception and redirection.

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 group communication systems such as Isis [1]. We believe that this goal is reached. The distributed diary application was initialy 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 [6]. Indeed, reflection in GARF is applied only to inter-object communication, and there is no recursivity in the model, i.e no meta-meta-objects.

Acknowledgements

The GARF project was funded by the Fonds National Suisse de la Recherche Scientifique as part of the Programme Prioritaire Informatique (SPP IF) under project number 5003-

034344. Many of our collegues have contributed in GARF, including X. Défago, R. Mattman, B. Muganga, F. Pacull, and A. Schiper.

References

[1] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press. 1993.

[2] F. Cristian. *Understanding Fault-Tolerant Distributed Systems.* In Communications of the ACM. February 1991.

[3] B. Garbinato, R. Guerraoui, and K. Mazouni. *Programming Fault-Tolerant Applications Using Two Orthogonal Object Levels.* In Proceedings of the International Symposium on Computer and Information Science. November 1993.

[4] B. Garbinato, R. Guerraoui, and K. Mazouni. *Distributed Programming In GARF.* In Object-Based Distributed Programming. Springer Verlag, LNCS 791. June 1994.

[5] 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.

[6] R. Guerraoui, B. Garbinato, and K. Mazouni. *Lessons from Designing and Implementing GARF.* In Object-Based Parallel and Distributed Computation. Springer Verlag, LNCS 1107. June 1996.

[7] V. Hadzilacos and S. Toueg. *Fault-Tolerant Broadcasts and Related Problems.* In Distributed Systems, Sape Mullender Editor, ACM Press Publisher. 1993.

[8] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Metaobject protocol.* The Mit Press. 1991.

[9] L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 21(7). July 1978.

Related work

Isis was the first system to introduce group communications for supporting reliable distributed applications [3]. Inspired by Isis, several group communication systems such as Transis and Totem [3] have recently been developed. These systems provide a set of inter-process communication primitives that enable to multicast messages to groups of processes. These primitives differ in the way messages are ordered (e.g total order/causal order) and in the way atomicity of message delivery is ensured in the case of process failures. Programming with group communication primitives requires a broad experience in distributed computing as one must be familiar with processes and inter-process communications (e.g call-backs). In addition, group communication primitives must be mixed with application functional features, hampering extensibility and modularity. GARF hides group communication primitives from the programmer and separates features that concern distribution and reliability, from functional aspects.

Systems like Arjuna [4] are similar to GARF in that they lie at an intermediate level between the programmer and group communication systems. Arjuna supports a high programming level (object level) and provide adequate abstractions for remote invocation, object replication and object group communication. Nevertheless, abstractions related to distribution and replication must be mixed with application functional features. Reflective systems like Maud [1] separate design concerns. In reflective systems, each object is bound to a group of *meta-objects* describing the object behavior. Each meta-object is also bound to a group of meta-objects, and so on. Although very flexible, reflective systems have complicated semantics and may bring unnecessary additional complexity. We believe that since the only way to access an object is to invoke one of its operations, it suffices to control the invocation process in order to express various behaviors. In fact, GARF encapsulators and mailers can be viewed as dedicated meta-objects, and GARF can be viewed as a pragmatic reflective tool. The most similar approach to ours is the one based on first class interactions, as in Sina [1]. In Sina, each object is associated with a set of *incoming filters* and *outcoming filters*. Filters may delay a message, redirect it or even change it. Since filters are locally associated to objects, they behave just like GARF encapsulators. No facilities (such as GARF mailers) are provided for remote communication and programmers can only act on what happen when the message arrives at the object server side. It is for instance impossible to compute a replicated object server in a transparent way from it clients.

References

[1] Agha et al. *A Linguistic Framework for Dynamic Composition of Dependability Protocols.* In Proceedings of the IFIP Conference on Dependable Computing for Critical Applications. Elsevier. 1993.

[2] Aksit et al. *Abstracting Inter-Object Communications Using Composition Filters.* In Object Based Distributed Programming. Springer Verlag, LNCS 791. 1993.

[3] D.Powell (guest editor). Communications of the ACM, 39(4). Special issue on Group Communication. April 1996.

[4] 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.

Status of GARF

GARF is the French acronym for "Génération Automatique d'applications Résistantes aux Fautes" (which means Automatic Generation of Reliable Applications). The objective of the GARF project (started in 1992), was to design and to implement a software that automatically generates a distributed reliable application from a centralized one (by replicating its critical components over several machines). The resulting software, also named GARF [2], has been implemented in Smalltalk-80 on top of the Isis [1] group communication system, and later ported on the Phoenix [3] system. The GARF runtime, in charge of associating data objects to encapsulators and mailers, has been implemented using the reflexive facilities of Smalltalk [4]. These associations 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. Encapsulator and mailer classes (the basic components that manage distribution and replication) are implemented using the Isis communication primitives. To avoid interference between Smalltalk-80 processes and Isis threads, A GARF/Isis interface is implemented inside a dedicated Unix process.

The GARF runtime introduces an overhead factor of 70 without considering distribution and reliability. A normal (neither encapsulator nor mailer are associated to objects) Smalltalk invocation of an object has a response time of $9\mu s$ (in a Sun SPARCstation equipped with 32 Mbytes RAM and running Solaris 2.2.). When intercepted by the GARF runtime and redirected to mailers and encapsulators, the 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, the virtual machine reinterprets the operation that caused the exception, which makes the execution much slower. The time spent in the GARF runtime and in data objects is however small compared to the $251ms$ that the same invocation takes for a remote object.

GARF is currently used in practical teaching of distributed computing at the Swiss Federal Institute of Technology in Lausanne. It is available without charge to researchers on demand at garf@lse.epfl.ch. More information on GARF can also be found at http://lsewww.epfl.ch/garf.

References

[1] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press. 1993.

[2] B. Garbinato, R. Guerraoui, and K. Mazouni. *Implementation of the GARF Replicated Objects Plateform.* In Distributed Systems Engineering Journal, (2). 1995.

[3] C. Malloth. *Conception and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, 1996.

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