Filtering Duplicated Invocations Using Symmetric Proxies*

Karim R. Mazouni Benoît Garbinato Rachid Guerraoui
Laboratoire de Systèmes d'Exploitation, Département d'Informatique
École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne (Switzerland)
phone: + 41 21 693 52 73 fax: + 41 21 693 67 70 e-mail: mazouni@lse.epfl.ch

Abstract

Invocation is the communication paradigm of objects. It is asymmetric by definition: the invoker initiates the invocation whereas the invokee reacts to it. A proxy is a local representative of the invokee on the invoker's machine. Like the invocation, the proxy notion is asymmetric. However, when replicated objects are considered, there is a need for symmetric proxies*: a regular proxy plus an invoker proxy on the invokee's machine.*

This paper shows that symmetric proxies are needed to implement pre-filtering *of* duplicated invocations. *It defines both the duplicated invocation problem and the pre-filtering approach as an efficient solution. The paper presents an object model based on symmetric proxies. Finally, the implementation of the model in the GARF system is presented as well as similar works in the litterature.*

1 Introduction

Modern computing is definitely distributed. More and more computers of every size (from laptops to mainframes) need to exchange information across a network. Hence, there is a growing need for techniques to build and to support distributed computing. In particular, fault tolerance is an important issue since more and more aspects of everyday life rely on computers. Fault tolerance is the ability of a system to remain *fully* operational (ie no loss in service quality) despite the failures of some of its components.

The object paradigm which appeared first in programming languages, is now a well accepted abstraction for building distributed systems. An essential reason of the success of object technology is that objects implement naturally the client-server model. The latter is the reference model for designing a distributed system. The client-server model defines a distributed system as a set of components which communicates using *request-reply* interactions.

Replication is the *only* means to achieve fault tolerance [10]. Hence, object-based distributed systems must provide support for replicated objects. According to the encapsulation property of objects, a replicated object should be invoked *as it were not* replicated. In other words, object replication should be transparent. Replication transparency is difficult to achieve because it encompasses both *plurality* transparency [3] and *replication policy* transparency. Plurality transparency means that a replicated object should appear as it were monolithic, although it is actually composed of a collection of replicas distributed among several computers. Replication policy transparency means that the replication policy should not modify both the way the object is invoked and the way the object invokes other objects.

A flexible way to achieve replication transparency is to hide both plurality and replication policies in the mechanisms implementing the invocation abstraction. The paper use this approach in order to solve the *duplicated invocation* problem. The latter occurs whenever *actively replicated* objects interact.

Section 2 describes both the duplicated invocation problem and a solution called the pre-filtering approach. Section 3 presents an object model based on *symmetric proxies* which are needed to support pre-filtering. Section 4 presents the implementation of symmetric proxies in the GARF system. Section 5 overviews related work and Section 6 concludes the paper.

*Research funded by the Fonds National Suisse de la Recherche Scientifique as part of the Programme Prioritaire Informatique (SPP IF) under project number 5003-034344.

2 Filtering duplicated invocations

This section presents briefly the *duplicated invocation* problem and the *pre-filtering* approach, a solution to this problem. More details about both the problem and the solution can be found in [8].

2.1 Duplicated invocations

Duplicated invocations occur whenever an *actively replicated* object invokes another object. Active replication [10] is a widely used [13, 11, 4] replication policy which enforces an *identical* behaviour on all replicas.

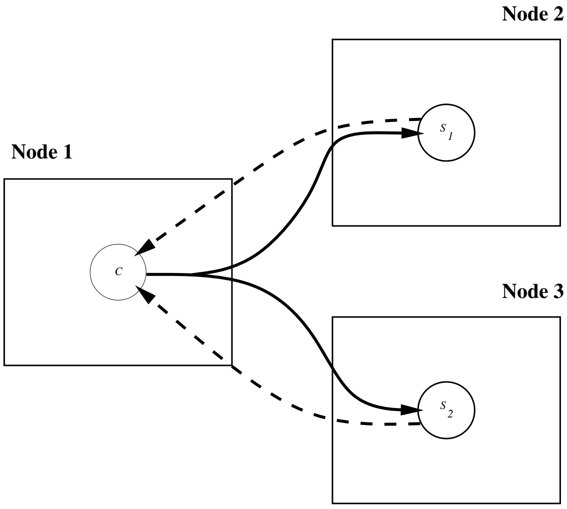

Figure 1: Invoking an actively replicated object

Figure 1 shows an example in which a non replicated object C invokes an actively replicated object S. The invoker C is on node 1 whereas the invokee S has two replicas S_1 and S_2, respectively on node 2 and on node 3. Every replica S_i receives the invocation request (plain curved arrows in Figure 1), performs the corresponding operation and returns a reply to C (dashed curved arrows). Note that the invoker C receives *multiple* replies to its single invocation. Hence, it should be able to deal with these *duplicated* replies: all replies are identical unless the replicas are not deterministic.

Suppose that the replicated object S invokes an actively replicated object P. This situation is depicted in Figure 2. The invokee P is replicated twice on node 2 and node 3. Every replica S_i sends an invocation request (plain arrows on Figure 2) to P's replicas. Each P_j receives and performs the same invocation twice instead of once: this is the *duplicated invocation* problem.

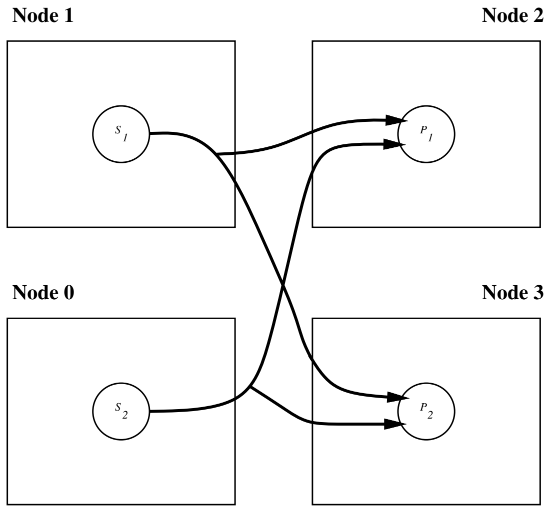

Figure 2: A duplicated invocation example

The risk of a duplicated invocation is the possible corruption of the invokee's state. If the operation corresponding to the invocation modifies the invokee's state in a non *idempotent* manner, the invokee's state is corrupted. An operation is idempotent if the results obtained after several executions of the operation is the same as those obtained after a single execution. For instance, an operation which increments a counter is not idempotent whereas an operation which resets the counter is idempotent.

2.2 Pre-filtering

A solution to the duplicated invocation problem is to *pre-filter* any invocation involving actively replicated objects in order to avoid duplicated invocations. Pre-filtering consists in filtering invocations at the invoker. The idea is to ensure that a single invoker's replica (called the *invoker's coordinator*) sends the invocation request across the network. Every invokee's replica receives the request and performs the corresponding operation. A single replica of the invokee (called the *invokee's coordinator*) sends a reply to the invoker's replicas.

Figure 3 shows an example of the pre-filtering approach. S_1 is the invoker's coordinator and P_2 is the invokee's coordinator. Pre-filtering saves computer resources (especially network bandwidth) because it actually *avoids* duplicated invocations. Replication masks failures of both the invoker and the invokee.

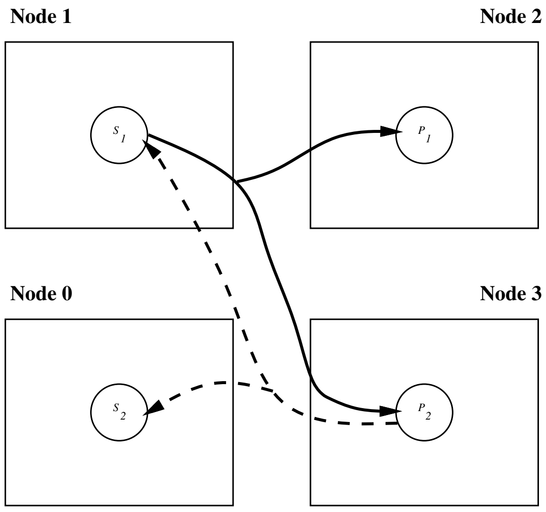

Figure 3: The pre-filtering approach

However, coordinator failure cases must be handled explicitly. Whenever the invoker's coordinator S_1 fails, the invocation request may never reach P's replicas. S replicas work this problem out by electing a new coordinator (S_2 in Figure 3) which sends a second invocation request to P's replicas. Since the new coordinator ignores whether the invocation is duplicated[1], it is up to P's replicas to detect duplicated invocations and to discard them. In this case, it is the invokee which filters duplicated invocations. The failure of the invokee's coordinator is solved using the same algorithm.

3 Symmetric proxies and filters

This section describes an object model based on symmetric proxies which supports replicated objects. In particular, the model supports pre-filtering of duplicated invocations. Section 3.1 overviews the model and defines the concept of *symmetric* representatives. Section 3.2 shows that representatives are actually extended proxies. Section 3.3 gives a full description of the model based on *proxies* and *filters*.

3.1 Symmetric representatives

Filtering duplicated invocations involves complex issues related to both distribution and replication. For instance, objects should be able to exchange messages across the network, locate each other replicas and so on. These issues are not only independent of application functionalities but they also complicate application programming. Hence, issues related to both distribution and replication should be addressed once for all and reused among applications.

A good approach is to hide distribution and replication from applications. In other words, distribution and replication are made transparent in the sense that objects interact as they were not distributed nor replicated. To achieve both distribution transparency and replication transparency, it is useful to consider an object model with two layers: *application* objects and *communication* objects. Application objects implement application functionalities without considering distribution and replication: they are application specific. In contrast, communication objects implement both distribution transparency and replication transparency on behalf of application objects.

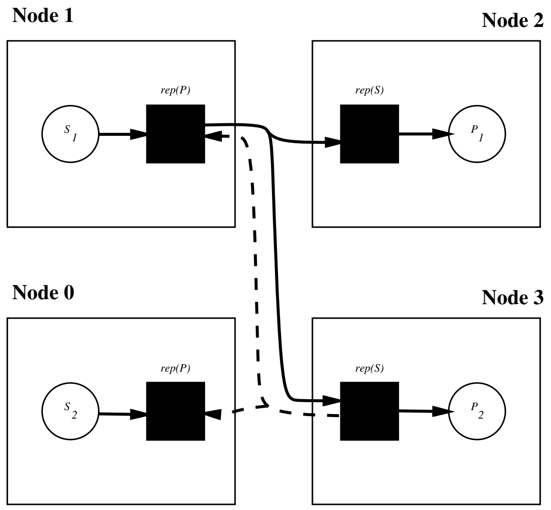

Figure 4: Symmetric representatives

Figure 4 presents the model using the example of Figure 3. An application object S invokes an application object P. Both the invoker and the invokee are actively replicated. Plain circles represent application objects. The latter communicate using local invocations (straight plain arrows in Figure 4). An invocation is composed of both a request and a reply. It is the standard communication means of objects. Black boxes represent communication objects. They communicate across the network using messages. A message may carry either an invocation request (plain curved arrow in Figure 4) or an invocation reply (dashed curved arrow).

Communication objects act as local *representa-*

[1]The first invocation might reached P's replicas despite the invoker failure.

tives of remote objects. On each invoker's node (ie node 0 and node 1), there is a representative of the invokee. In Figure 4, each replica S_i actually invokes a P's representative $rep(P)$. Symmetrically, there is a representative of the invoker on each invokee's node (ie node 2 and node 3). It is actually a S's representative $rep(S)$ which invokes P_j.

Application objects are not aware of both distribution and replication. They rely completely on communication objects for these issues. In particular, communication objects solve duplicated invocations using pre-filtering. In Figure 4, each replica S_i invokes P as it were a local object. Actually, S_i invokes $rep(P)$. Since $rep(P)$ is aware of P's replication policy, it is able to locate P's replicas. Furthermore, P's representatives are able to elect an invoker's coordinator ($rep(P)$ on node 1) which actually forwards the invocation request to S's nodes.

On the invokee's nodes, $rep(S)$ receives the invocation request and invokes the local replica P_j. Each $rep(S)$ eventually obtains from P_j the result of the invocation. S's representatives elect an invokee's coordinator ($rep(S)$ on node 3) which forwards the reply to the invoker's nodes. Each $rep(S)$ receives the invocation reply and forwards it to S_i.

The network communication performed by representatives is transparent for application objects. On the invoker's side, each S_i performs a *local* invocation on P (actually on a representative $rep(P)$) and eventually gets a reply when the invocation returns. On the invokee's side, each P_j receives a regular *local* invocation, performs the corresponding operation and returns the result.

3.2 Representatives are extended proxies

On the one hand, representatives are similar to proxies. From an application object's point of view, a representative is indistinguishable from the remote object it represents. According to Shapiro's definition in [12], a *proxy* provides a client object (ie an invoker) with a local interface to a *group* of remote objects (ie invokees) called *principals*. In this model, a distributed object is composed of both the principals and the proxy. The client invokes locally the proxy which communicates with the principals across the network. The proxy gives the illusion to the client that the distributed object is local.

On the other hand, representatives differs from proxies on two main aspects:

- representatives are symmetric whereas proxies are not;
- representatives' behaviours depend on both the invoker and the invokee whereas proxies' depends only on the invokee.

Symmetric representatives vs asymmetric proxies. Representatives are *symmetric*: there are both an invokee's representative on the invoker's node and an invoker's representative on the invokee's node. In contrast, Shapiro's model is *asymmetric*: a proxy represents the invokee on the invoker's node. The symmetry of representatives provides more flexibility because it permits a better control over the actual network commmunication. In particular, the message carrying an invocation reply may be completely *independent* of the message carrying the request.

In this context, independence means that the communication path of the reply may be different from the *reverse* request path. For instance in Figure 4, a request is sent from node 1 to both node 2 and node 3, whereas the reply is sent from node 3 to both node 0 and node 1. The motivation of such communication patterns is to hide replication of both the invoker and the invokee. The communication pattern of Figure 4 may be interpreted as "sending the request to P's replicas" followed by "sending the reply to S's replicas". Representatives are aware of the replication policies and they perform the corresponding communication.

This level of flexibility cannot be achieved using regular proxies. Proxy asymmetry comes from the invocation asymmetry: the proxy model extends the invocation paradigm to distributed objects. An invocation is intrinsically asymmetric because the invoker *needs* to know the invokee's name whereas the invokee needs not. The proxy actually represents the invokee's name on the invoker's node. Since the invoker cannot perceive what is beyond this invokee's name, the proxy encapsulates the real invokee, hiding both distribution and replication from the invoker. In contrast, the invokee is somewhat aware of distribution: it is able to communicate with a remote proxy across the network. Furthermore, the communication pattern is *fixed*: the proxy sends a request message to the invokee, and later the reply is sent from the invokee to the proxy. The reply follows *exactly* the reverse request path.

Representatives' behaviours vs proxies' behaviours. The behaviour of a proxy depends only on the object which the proxy represents, ie the invokee. In contrast, the behaviour of a representative depends on both the invoker and the invokee. For instance in Figure 4, each $rep(P)$ depends not only on P but also on S. Representatives of P perform pre-filtering to avoid duplicated requests *because* S is actively replicated. If S were not replicated, no duplicated requests would occur. Similarly, S representatives pre-filter duplicated replies caused by the active replication of P.

In order to respect the encapsulation property of objects, the two parts of representative behaviour should be separated and controlled by different communication objects. This leads to a model where a representative is actually composed of two specific communication objects: a *proxy* and a *filter*.

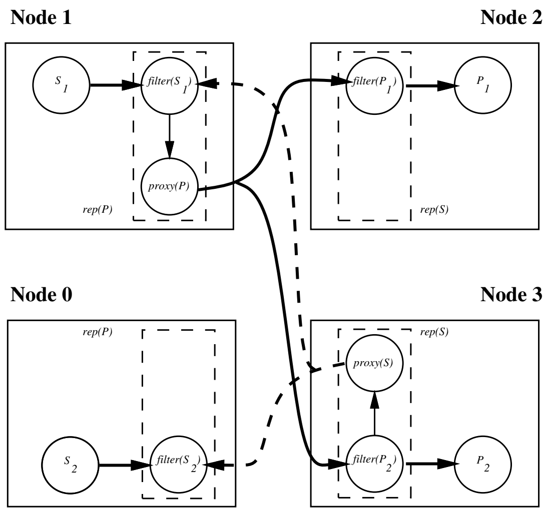

Figure 5: Symmetric proxies and filters

3.3 Filters and symmetric proxies

Figure 5 presents the model using proxies and filters to implement representatives. Each representative (dashed boxes) is composed of both a proxy representing the remote object and a filter representing the local object. For instance, $rep(P)$ on node 1 is composed of the proxy $proxy(P)$ and the filter $filter(S_1)$. Symmetrically, $rep(S)$ on node 3 is composed of both $proxy(S)$ and $filter(P_2)$.

A filter is associated to each application object. It controls the communication between the application object and the external world. Each invocation concerning the application object, either as invoker or as invokee, passes through the filter. The latter receives incoming invocations packed in network messages sent by remote proxies. For instance, $filter(P_1)$ and $filter(P_2)$ receive the invocation request sent by $proxy(P)$ from node 1. To send outgoing invocations, a filter uses a local proxy of the invokee.

A proxy is created whenever a message should be sent to the object it represents. The message could carry either a request or a reply. For instance, a proxy of P noted $proxy(P)$ is created on node 1 to send the invocation request to P's replicas. Symmetrically, a proxy of S noted $proxy(S)$ is created on node 3 to send the invocation reply to S's replicas.

4 Implementing symmetric proxies in the GARF system

This section presents the implementation in the GARF system of the object model described in Section 3. Section 4.1 overviews the GARF system. Section 4.2 describes the object model of GARF.

4.1 Overview of the GARF system

GARF is an object-oriented environment which aims to facilitate the programming of fault-tolerant distributed applications. GARF achieves this goal by proposing a programming model composed of two object layers: the *application* layer and the *communication* layer.

Programmers build fault-tolerant distributed applications in two steps. First, they program application objects. Application objects interact using regular invocations ignoring everything about distribution and replication. Second, programmers associate critical application objects with appropriate communication objects picked up from the GARF library. Communication objects have the ability to send and to receive network messages. Their role is to achieve both distribution transparency and replication transparency on behalf of application objects.

Both layers are actually made of regular Smalltalk objects in the current implementation. Communication objects rely on specific software (TCP/IP and ISIS [2]) to send and to receive messages across the network. On each node (see Figure 6), the GARF system is implemented using two UNIX processes: **GARF/ST** and **DOT/C**. The process **GARF/ST** runs a regular Smalltalk image which contains the core of the GARF system. The process **DOT/C** implements an interface to the ISIS toolkit and communicates with **GARF/ST** using UNIX sockets.

Communication objects implement replicated objects using object groups. The latter are mapped on UNIX process groups provided by ISIS. The ISIS toolkit implements UNIX process group management and group multicast facilities. In Figure 6, S's replicas on node 0 and on node 1 are members of an ob-

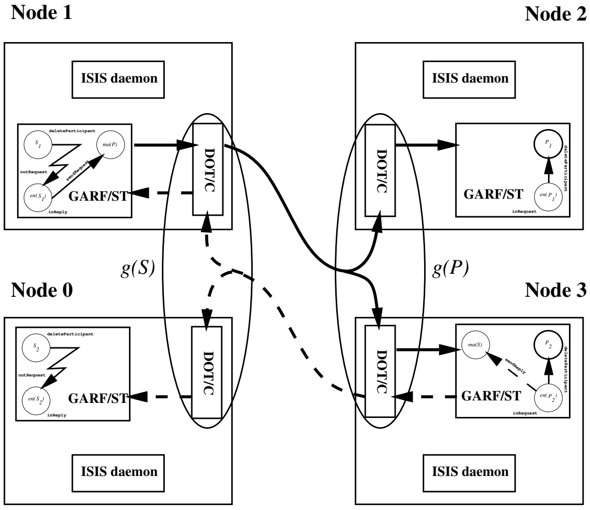

Figure 6: Architecture of GARF

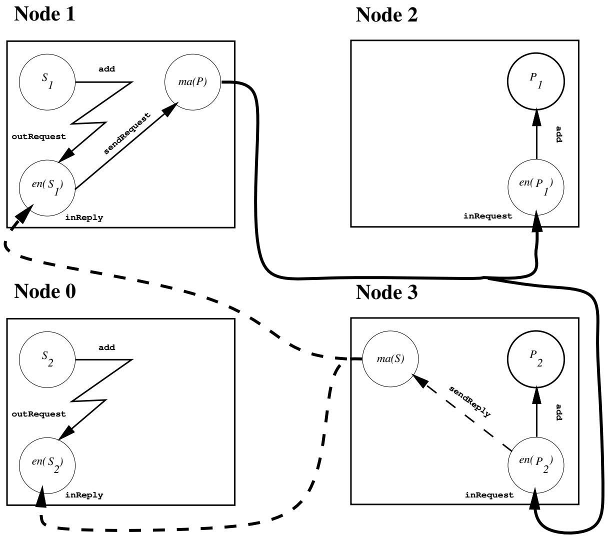

Figure 7: Pre-filtering in the GARF model

ject group. The latter is implemented using a process group $g(S)$ whose members are the **DOT/C** processes of node 0 and node 1. Similarly, the process group $g(P)$ implements the group of P's replicas.

4.2 GARF's object model

According to the model presented in Section 3.3, there are two kinds of communication objects in the GARF system: *encapsulators* and *mailers*. Encapsulators implement the filter behaviour whereas mailers implement the proxy behaviour. Each application object O which should be distributed or replicated is associated with two communication objects: an encapsulator $en(O)$, and a mailer $ma(O)$.

The encapsulator $en(O)$ controls *every* invocation in which O is involved[2]. If O is the invoker, $en(O)$ performs actions before sending out the request to the invokee, or before transmitting the reply to O. If O is the invokee, $en(O)$ performs actions before transmitting the request to O, or before sending out the reply to the invoker. These actions are part of the three operations that every encapsulator implements: **outRequest**, **inRequest** and **inReply**. An encapsulator is an *invocation filter* which is always on the node where its associated application object resides.

In contrast, the mailer $ma(O)$ is never on O's node: it resides on a remote node and carries either requests or replies from the remote node towards O's node. A mailer may be viewed as an extended proxy. If O is the invokee, $ma(O)$ behaves like a regular proxy: it represents the invokee on the invoker's node and forwards, across the network, the invocation request to O. If O is the invoker, $ma(O)$ represents the invoker on the invokee's node and forwards, across the network, the invocation reply to O. These actions are part of the two operations which every mailer implements: **sendRequest** and **sendReply**.

Figure 7 uses the example of Figure 5 to show how pre-filtering is expressed in the GARF model. The two figures are very similar. The only difference is that Figure 7 shows the mechanisms used to hide the communication layer from the application layer. The "lightning" like arrows represent *redirected* invocations. Whenever an application object S_i invokes an application object P, the invocation is intercepted at runtime by the GARF system which redirects it towards S_i's encapsulator en(S_i). As a result, the communication layer is completely transparent for the application layer.

Implementation of invocation pre-filtering in the GARF system. Figure 7 gives details about the implementation of invocation pre-filtering in the GARF system. Whenever S_i invokes P using a regular invocation statement, the invocation request is intercepted by the GARF runtime which invokes the operation **outRequest** of S_i's encapsulator $en(S_i)$.

The operation **outRequest** builds a unique identifier for the request. This identifier is needed to associate a request with its reply because each is perceived as a separate event by $en(S_i)$. Every $en(S_i)$ belongs to

[2]Either as invoker or as invokee.

the same object group. Group members are aware of the group membership and they are able to elect deterministically the same coordinator. The invoker's coordinator $en(S_1)$ transmits the request to the communication object which represents P, the mailer $ma(P)$. The identifier is appended to the request and passed to the mailer through the invocation of its operation **sendRequest**. The role of **sendRequest** is to send the invocation request across the network.

The mailer $ma(P)$ packs the request in a network message and sends it to every $en(P_j)$ on the invokee's nodes. To perform this task, $ma(P)$ relies on distributed services provided by additional software (cf Section 4.1). In particular, mailers uses a *name service* which manages *location independent identifiers* for encapsulators. Since every distributed application object has an encapsulator, giving identifiers to application objects is useless.

After sending the request across the network, **sendRequest** completes and the control is given back to the calling operation **outRequest**. The latter should not terminate before the invocation reply arrives from the invokee P. The reply is delivered to $en(S_i)$ through a call of its operation **inReply**. Thus, **outRequest** must wait for the reply *explicitly*, ie using some synchronization mechanism. The synchronization is needed because the invocation abstraction used by the application layer is implemented by the communication layer using *two* one-way network messages carrying respectively the request and the reply. The operation **outRequest** waits for the reply using a private semaphore[3].

On the invokee's nodes, the delivery of the network message carrying the request is performed by the runtime. The latter invokes the operation **inRequest** of $en(P_j)$ passing the request as an argument. The encapsulator $en(P_j)$ unpacks the request storing its identifier before actually invoking P_j. When P_j returns the operation result to $en(P_j)$, the latter builds a reply composed of both the result and the request identifier.

Every $en(P_j)$ belongs to the same object group and elects the same invokee's coordinator. The latter $en(P_2)$ transmits the reply to the invoker's mailer $ma(S)$ invoking its **sendReply** operation. The mailer $ma(S)$ uses the same mechanisms as $ma(P)$ to forward the reply to the invoker's encapsulator $en(S_i)$. The reply delivery is performed by the operation **inReply** of $en(S_i)$. The invocation result and the identifier are extracted from the reply. The identifier is used for retrieving the private semaphore in order to resume **outRequest** execution. As soon as **outRequest** completes, the invoker object S_i obtains the invocation result.

[3]Semaphores are provided by Smalltalk (Class *Semaphore*)

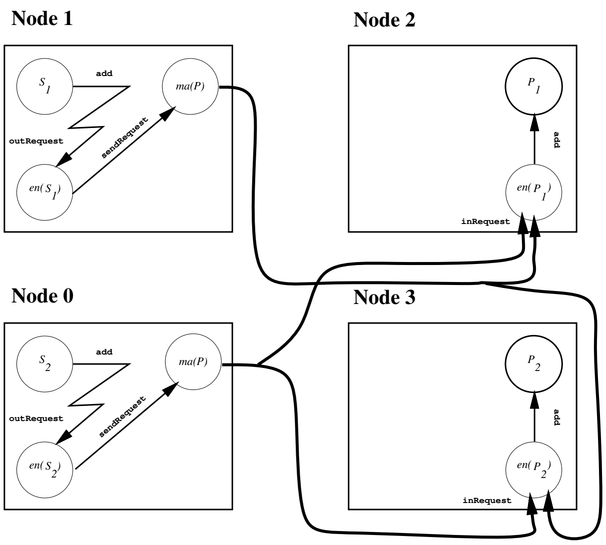

Figure 8: Impossibility of pre-filtering in the GARF's initial model

The initial object model of the GARF system. The model described in this paper is an update of the initial model described in [5]. The terminology was slightly different: application objects were called *data* objects and communication objects were called *behavioural* objects.

The main difference between the two models resides in the pattern of network messages implementing an invocation. The initial model was based on asymmetric proxies: there was no mailer on the invokee's site to send the reply. Figure 8 shows that pre-filtering was impossible using the initial object model of GARF. The problem is that the reply followed systematically the reverse request path.

5 Related work

Several object-based distributed systems like *GAGGLES* [3], *XENOOPS* [1], *SOR* [6], *ARJUNA* [14] and ANSA [9], provide support for replicated object invocations. However, only a few authors explicitly describe how invocations of replicated objects are implemented. In particular, it is difficult to find out how problems like duplicated invocations are solved.

GAGGLES is a lightweight mechanism intended to encapsulate plurality. It provides support for invoking a gaggle of objects as if they were a single object. *GAGGLES* alone cannot implement replicated objects. According to its authors, it may be combined with a mechanism implementing replication in order to support replicated objects.

XENOOPS is an object oriented environment built as an extension of C++. Its *Object Invocation Subsystem* supports invocations of replicated objects using a hierarchy of reference classes. For instance, an actively replicated object is referenced using a instance of class *ActiveReplicatedReference*. Whenever an actively replicated object is invoked, its reference object intercepts the invocation and forwards it as a reified object to every replica. The invoker is not aware of the invokee replication: the *Object Invocation Subsystem* hides invokee replication. This approach is interesting but *XENOOPS* authors do not mention what happens if both the invoker and the invokee are replicated.

The *SOR* project proposes to implement replicated objects using the *fragmented object* model. A fragmented object is capable of implementing several replication policies. However, it is not clear how several fragmented objects interact and how problems like duplicated invocations are avoided.

ARJUNA implement replicated objects using group of replicas. Several C++ classes implement different replication policies. Stub mechanisms hide replication details from the programmer. These mechanisms include group management and group communication (reliable group RPC). According to the authors, the group RPC ensures that a duplicated invocation behaves like a non-duplicated one. Although it is not clearly stated, duplicated invocations seem to be filtered at the invokee [7] using invocation identifiers.

The designers of *ANSA* propose to implement a replicated object using an *interface group*. An interface group is logically composed of *member agents*, a *distributor* and a *collator*. Each replica is associated with a single member agent. The latter hides replication from the replica. The distributor distributes invocation requests to member agents according to the replication policy. Each member agent invokes its replica discarding duplicated invocations (ie invocation filtering at the invokee). The collator collects invocation replies from member agents and returns a single result to the invoker. In order to avoid that both the distributor and the collator become single points of failures, they are replicated and collocated with each invoker. However, it is not clear how this solution works when both the invoker and the invokee are replicated.

6 Conclusion

This paper showed that an object model based on symmetric proxies is needed to implement invocation *pre-filtering*. Pre-filtering is a technique which avoids *duplicated invocations*. Section 2 presented the duplicated invocation problem and the pre-filtering approach. Section 3 showed that pre-filtering is elegantly implemented using *symmetric representatives*. The latter are a sort of extended proxies which implement both distribution transparency and replication transparency on behalf of application objects. The symmetric representative model permits better flexibility than the asymmetric proxy model. However, the latter can be used directly at the application layer whereas the former is part of the communication layer. The application layer is composed of application objects which are application specific. In contrast, the communication layer is made of communication objects which may be used from an application to the other. Communication objects implement representatives using filters and symmetric proxies. Section 4 showed how symmetric proxies are implemented in the GARF system. Finally, Section 5 looked at some object-based systems which support replicated objects.

References

[1] 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, ECOOP'93 Workshop Proceedings*, number 791 in Lecture Notes in Computer Science, pages 138–151. Springer-Verlag, 1994.

[2] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[3] A.P. Black and M.P. Immel. Encapsulating plurality. In Oscar M. Niestraz, editor, *ECOOP'93-Object-Oriented Programming, 7th European Conference*, number 707 in Lecture Notes in Computer Science, pages 56–79. Springer-Verlag, 1993.

[4] E.C. Cooper. Replicated procedure call. *ACM Operating Systems Review*, 20(1):44–56, January 1986.

[5] 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*, number 901 in Lecture Notes in Computer Science, pages 225–239. Springer Verlag, 1994.

[6] Y. Gourhant. An object-oriented approach for replication management. In *Proceedings of the 2nd Workshop on the Management of Replicated Data*, pages 74–77. IEEE Computer Society Press, November 1992.

[7] M.C. Little. *Object Replication in a Distributed System*. PhD thesis, University of Newcastle-upon-Tyne (UK), September 1991.

[8] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Building reliable client-server software using actively replicated objects. In I. Graham, B. Magnusson, B. Meyer, and J.-M. Nerson, editors, *Technology of Object-Oriented Languages and Systems, TOOLS 16*, pages 37–51. Prentice-Hall, 1995.

[9] M.H. Olsen, E. Oskiewicz, and J.P. Warne. A model for interface groups. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 98–107, October 1991.

[10] D. Powell, editor. *Delta4: A Generic Architecture for Dependable Computing*. Springer-Verlag, 1991.

[11] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[12] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE Computer Society Press, May 1986.

[13] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, pages 66–73, 1991.

[14] S.K. Shrivastava and D.L. McCue. Structuring fault-tolerant object systems for modularity in a distributed environment. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):421–432, April 1994.