Building Reliable Client-Server Software Using Actively Replicated Objects*

Karim R. Mazouni **Benoît Garbinato**
Rachid Guerraoui
Laboratoire de Systèmes d'Exploitation, Département d'Informatique
École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne (Switzerland)
e-mail: garf@lse.epfl.ch

Abstract

Combining replication and objects is a profitable approach to build fault tolerant distributed software. However, replication brings up several problems related to object interactions. In particular, this paper investigates the *duplicated interaction* problem which is raised whenever actively replicated objects interact. The paper presents the *pre-filtering* approach, a general solution to this problem. Pre-filtering has been implemented in GARF, an object-oriented system intended to simplify the programming of fault tolerant distributed software.

1 Introduction

Replicated objects are profitable to implement fault tolerant distributed software because of two essential reasons. First, replication is a well known and widespread technique used to achieve fault tolerance [Cristian 91]. Second, objects implement naturally the *client-server* model which is the basic paradigm for programming distributed software. Object replication generalizes the client-server model in order to support fault tolerance. In this generalized model, both clients and servers are replicated because objects are alternatively clients (ie invoking other objects) and servers (ie invoked by other objects). However, object replication policies interfer with each other and complicate interactions of replicated objects. Experiments made with active replication in the GARF system [Garbinato 93, Garbinato 94, Mazouni 94] raised the *duplicated interaction* problem. Object interactions are duplicated[1] in the sense that they are performed more times than needed, causing possible inconsistencies in object states.

This paper focuses on two instances of the duplicated interaction problem, viz *duplicated invocations* and *duplicated creations*. It presents the *pre-filtering* approach, a generic solution to both instances. Section 2 describes duplicated invocations and duplicated creations using examples from the *Distributed Diary Manager* (DDM), a fault tolerant distributed application implemented using the GARF system. Section 3 defines the pre-filtering approach. Section 4 presents an implementation of this solution in the GARF system. Section 5 glances the literature and Section 6 concludes the paper.

2 Active replication duplicates object interactions

Active replication [Powell 91] is a widely used [Shrivastava 91, Schneider 90, Cooper 86] replication policy which enforces an identical behaviour on all replicas. Each time a client object invokes an actively replicated server object, *each*

*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.

[1] The verb "to duplicate" is used along the paper in its second (negative) meaning ie "to do something again unecessarily". In contrast, the verb "to replicate"is used in its usual (positive) meaning ie "to make a copy of something".

server replica receives the invocation, performs the requested operation and replies to the client. There is no coordination among the replicas which behave as they were independent objects.

Hence, the client receives several replies to its invocation although only one is expected. Furthermore, assume that the replicated server interacts with a third party object to perform the requested operation. Then, the third party performs the same interaction several times instead of once. The reason is that each replica actually interacts with the third party. As a result, active replication causes object interactions to be *duplicated*: they are performed more than once and they may cause inconsistencies in object states.

This section describes two cases of duplicated interactions: *duplicated invocations* and *duplicated creations*. Each case is described using an example involving the Distributed Diary Manager (DDM). However, both cases are general and happen whenever active replication is used. DDM is a fault-tolerant distributed application designed in order to experiment with the GARF system. This application manages a replicated diary for each user. A diary contains the meetings to which the diary owner is expected to attend. A meeting is composed of a period (start date, end date) and a participant list. The application provides its users with several functionalities such as meeting scheduling and meeting cancelation.

2.1 Duplicated invocations

Whenever an actively replicated object invokes another object[2], the invocation is *duplicated* ie performed *more than once*, leading to possible inconsistencies in object states. This problem appears in the context of an invocation chain[3] containing at least an actively replicated object as an invoker.

For example, a duplicated invocation takes place each time a DDM user wishes to cancel a meeting. The semantics of meeting cancelation is as follows: whenever a user cancels a meeting, DDM removes this user from the meeting participant list[4]. Several objects are involved in a meeting cancelation operation (see Figure 1): an instance of the user interface U (class DiaryService), the user's diary D (instance of class Diary) and the corresponding meeting M (instance of class Meeting). Both D and M are actively replicated.

Figure 1 shows a case where D has 2 replicas (D_1 and D_2) and M has 3 replicas (M_1, M_2 and M_3). The meeting cancelation is performed in two steps: first, U invokes the method **cancelMeeting:** on D passing M as an argument; then D invokes the method **deleteParticipant:** on M. Since D is actively replicated, each D_i performs the invocation on M. As a result, each M_j receives and performs the invocation of **deleteParticipant:** twice (once per each D replica) instead of once. At each M_j, the first invocation succeeds but the next one fails because it attempts to remove a participant which no more belongs to the meeting list.

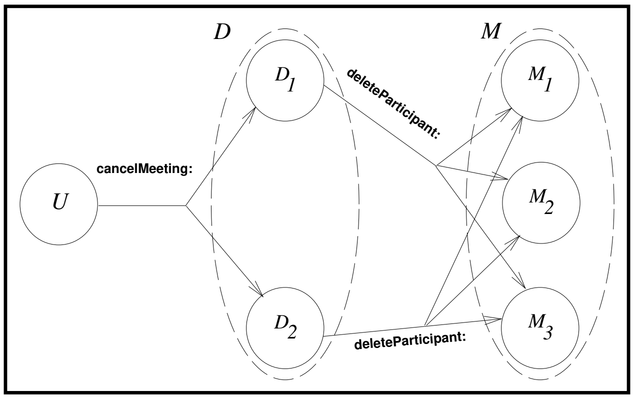

Figure 1: **A duplicated invocation in the context of DDM**

A duplicated invocation *does not corrupt* object states if all the operations performed in the invocation chain are *idempotent*: the result of several executions of an idempotent operation is equivalent to the result of a single execution. Any method which does not modify the state of an object is idempotent (eg a read operation). However, a duplicated invocation is always expensive in terms of computer resources (cpu, network bandwidth) regardless of the effects on the object states. Assume that the diary D is n-replicated[5] and that the meeting M is k-replicated. Each invocation of **cancelMeeting:** on D is actually performed n times (once per D_i). But, every D_i invokes **deleteParticipant:** on every M_j (because M is actively replicated too). As a result,

[2] either replicated or not

[3] An invocation chain is a sequence of nested invocations.

[4] If the list becomes empty the meeting is automatically deleted.

[5] D has n replicas in order to tolerate $n-1$ crash failures [Cristian 91].

the method **deleteParticipant**: is invoked n times on each M_j instead of once. Only k invocations (one per M_j) over $n \times k$ are needed: in other words $k(n-1)$ invocations are not only useless but they may corrupt the state of M. In the case of Figure 1, since $n = 2$ and $k = 3$ the method **deleteParticipant**: is invoked 6 times instead of 3.

2.2 Duplicated creations

Whenever an actively replicated object creates a *shared* object, the creation is *duplicated*, ie each replica creates its own object instead of creating a single object shared by all the replicas. For example, a duplicated creation takes place each time a DDM user wishes to schedule a meeting. Scheduling a meeting is performed in three steps: first, the participant's diaries are checked up to find a common free period, then the meeting is actually created and at last, the new meeting is scheduled in every participant's diary.

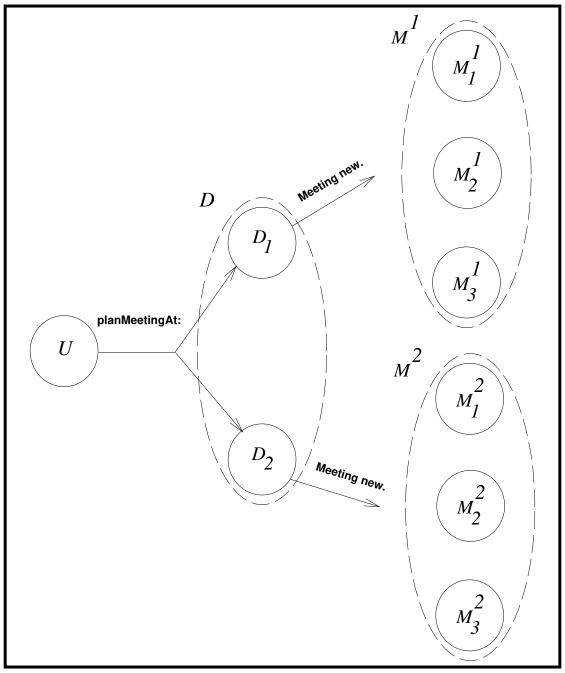

Figure 2: **A duplicated creation of a shared object in the context of DDM**

The second step ie meeting creation, involves an instance of the user interface U and the user's replicated diary D (see Figure 2). The object U invokes method **planMeetingAt**: on D. Each replica D_i then invokes[6] the class Meeting to actually create a meeting M^i. As a result, each D_i has created its own meeting instead of sharing the same object. Figure 2 shows that D_1 created M^1 while D_2 created M^2. Since meetings are replicated, each M^i is actually a replicated object. If D is n-replicated and the replication rate of every M^i is k, $n \times k$ copies of the same meeting are created although only k were expected.

If the created object is *private*[7] the situation depicted above is correct: several creations *are* expected if the creator is actively replicated. For example, whenever a new meeting is scheduled, the meeting object, say M, creates an instance P of class Period which contains the period of the meeting. The object P is private to its creator M: no other object has a reference on P. If M is actively replicated, every replica M_j creates its own period P^j. Figure 3 shows a situation where three replicas M_1, M_2 and M_3 have respectively created their own periods P^1, P^2 and P^3.

2.3 Discussion

Whether an object creation is duplicated or not, depends on the sharing state of the created object. This information is given by both the application semantics and the object dependencies resulting from the application design. For instance, the fact that DDM meetings are shared while periods are not, results only from design choices. One could imagine an alternative design where both meetings and periods are shared objects. Information on the sharing state of created objects is used by the system to detect duplicated creations in order to treat them correctly, ie as particular cases of duplicated invocations.

A duplicated creation is a sort of duplicated invocation which effects are external to the invokee: a new object is added to the system. Their semantics are similar in the sense that both the invocation and the creation must be performed only once. The creation result is a reference on the newly created object that the invokee returns to its invoker. Since duplicated creations can be detected by the system, they are solved by the same mechanisms which treat duplicated invocations. The protocol presented in the next

[6]In Smalltalk [Goldberg 83], the class method **new** is used to create objects.

[7]An object is private if its creator is the *only* object which has a reference on it.

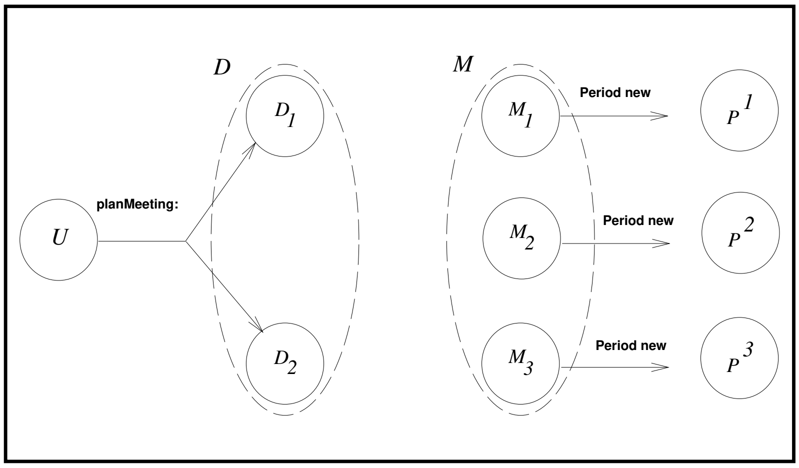

Figure 3: **A normal creation of a private object in the context of DDM**

section avoids duplicated interactions ie both duplicated invocations and duplicated creations. However, it is described only in the context of duplicated invocations.

3 Pre-filtering duplicated invocations

A solution to the duplicated invocation problem is to filter any invocation involving actively replicated objects in order to discard duplicated invocations. Two approaches are possible: *pre-filtering* and *post-filtering*. Pre-filtering consists in filtering invocations at the invoker while post-filtering consists in filtering invocations at the invokee. Consider the example shown on Figure 1. The invocation of **deleteParticipant**: on M by D is duplicated. If post-filtering is used, each D_i invokes M. Every M_j performs the invocation once, discarding the duplicates and replies to D. Every D_i discards all the duplicated replies but one. If pre-filtering is used, *only* one D_i invokes M. Every M_j receives a single invocation request and performs the request but *only* one replies to D.

Pre-filtering has an obvious advantage over post-filtering: it saves computer resources (especially network bandwith) because it actually *avoids* duplicated invocations. However, failure treatment is more complex with pre-filtering than with post-filtering. If the latter is used, no special treatment is needed for failure cases. If D_i fails, its invocation may never reach M. Since the invocation is duplicated M has several chances (ie up to the count of surviving D replicas) to receive it. If M_j fails, it may never perform the invocation. However, the invocation have several chances to be performed by the surviving replicas of M.

Pre-filtering has to handle failure cases explicitly. Whenever D_i invokes M and fails, the invocation may never reach M. D replicas work this problem out by electing another D_i which invokes M. Since D duplicas ignore whether the invocation is duplicated[8], it is up to M replicas to detect duplicated invocations and to discard them. In this case, it is the invokee which filters invocations as in post-filtering. The same algorithm is performed by M replicas to send the invocation reply to D. Whenever M_j replies to D and fails, the invocation reply may never reach D. M replicas elect another M_j which replies to D. For the same reasons as above, it is up to D replicas to discard duplicated replies.

This section presents a protocol which implements pre-filtering. It starts with an overview of the protocol (cf Section 3.1), presents the protocol algorithms (cf Section 3.2) and ends with the description of failure cases (cf Section 3.3).

3.1 Protocol overview

The protocol presented below is an implementation of the pre-filtering approach. The underlying system model consists in interconnected computers called

[8]The first invocation might reached M despite the invoker failure.

nodes where objects reside. A node may crash causing the failure of every object residing on the node. For each object O, there is a communication dedicated object called $Com(O)$ which actually performs the protocol algorithm on behalf of O. Figure 4 shows 3 objects C, S, P residing on 5 nodes. More precisely, C is not replicated and resides on node 1; S has two replicas S_1 and S_2 residing respectively on node 2 and node 5; P has two replicas P_1 and P_2 residing respectively on node 3 and node 4. C invokes S which in turn invokes P. Since S is actively replicated, the invocation of P by S would be duplicated if the protocol did not avoid it.

The protocol works as follows: whenever C invokes S, the invocation request is transmitted to $Com(C)$[9] which generates a unique identifier i for it and forwards the request (plain arrow 1 on Figure 4) to the nodes where S replicas reside. Each S_k receives request i via $Com(S_k)$ and invokes P. Each $Com(S_k)$ receives the invocation request of P and generates a unique identifier j for it[10]. At this point, a unique $Com(S_{coord})$ object (it is $Com(S_1)$ on Figure 4), called the *coordinator*, forwards request j (plain arrow 2) to every P_k's node. The other $Com(S_k)$ objects (ie $Com(S_2)$), called the *cohort*, store the request but *do not* forward it: this is *pre-filtering*.

Each P_k receives request j via $Com(P_k)$, performs it and generates a reply. The latter is transmitted to $Com(P_k)$ which identifies it as reply j, the reply to request j. At this point, the coordinator $Com(P_{coord})$ (ie $Com(P_1)$), sends reply j (dashed arrow 3) to S replicas. The cohort $Com(P_2)$ stores the reply but *does not* send it. Every S_k receives reply j via $Com(S_k)$ and builds reply i. The latter is transmitted to $Com(S_k)$ but only the coordinator $Com(S_1)$ forwards it to C via $Com(C)$. The cohort $Com(S_2)$ stores reply i silently.

3.2 Algorithms

Algorithms are written in pseudo-code from the point of view of an object S invoked by C and invoking P. However, they apply for any actively replicated object involved in a invocation chain (eg P).

3.2.1 Coordinator

The algorithm performed by the coordinator $Com(S_{coord})$ is shown in Appendix A. There are five events to which $Com(S_{coord})$ reacts:

1. $RECEIVE(request(i, C, S))$: $Com(S_{coord})$ receives request i sent by C to S;
2. $GET(reply(i, C, S))$: $Com(S_{coord})$ gets from S_{coord} a reply to request i;
3. $GET(request(j, S, P))$: $Com(S_{coord})$ gets from S_{coord} request j which destination is P;
4. $RECEIVE(reply(j, S, P))$: $Com(S_{coord})$ receives a reply to request j from P;
5. $ELECTION(Com(S_{coord}))$: $Com(S_{coord})$ is elected as the new coordinator because the previous coordinator failed (cf Section 3.3).

Whenever the coordinator receives request i from C (cf event $RECEIVE(request(i, C, S))$ in Appendix A) it checks if the request is already in the $Incoming$ table. Each entry in $Incoming$ stores information about an invocation received from a client object. This information is:

- id: a unique request identifier (eg (i, C, S) identifies request i sent by C to S);
- $status$: a value from the set $\{DONE, PENDING\}$ indicating whether a reply for this request has been received;
- $reply$: the actual reply if $status$ value is $DONE$.

If request i is not in $Incoming$, a new entry is allocated and the request is delivered to S_{coord}. Otherwise, the request has been already received and if a reply is available it is returned to the invoker; if not the request is discarded to avoid a duplicated invocation.

In response to a request, S_{coord} may generate either a reply or a new request to invoke another object. If $Com(S_{coord})$ gets a reply to request i (event $GET(reply(i, C, S))$), the coordinator stores

[9] On Figure 4, thin arrows represent object invocations (ie request + reply) and thick arrows represent network messages carrying either a request or a reply.

[10] The replicas are *deterministic*: they receives the same sequence of inputs and generate the same set of outputs. Thus, every $Com(S_k)$ generates the same identifier for a given request.

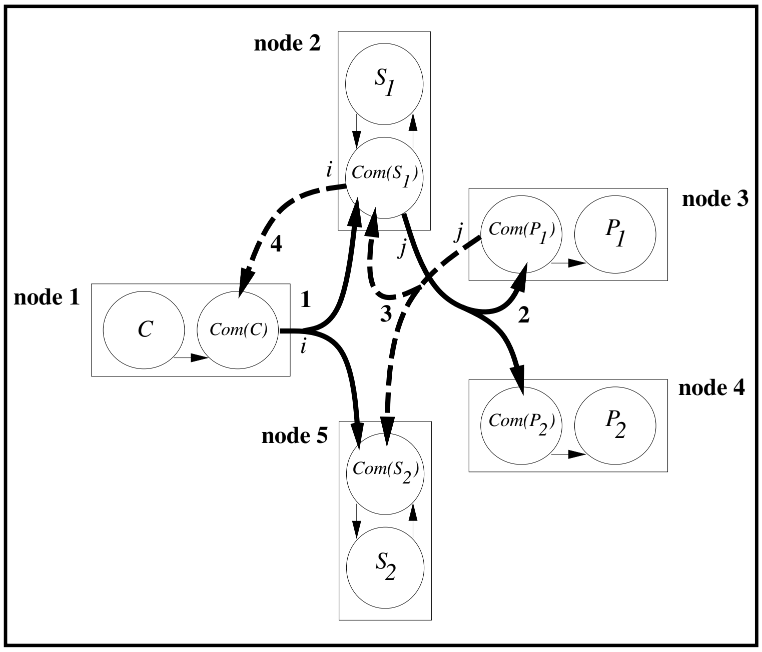

Figure 4: **A protocol for pre-filtering invocations**

the reply in $Incoming$ and sends it to its destination ie C. If $Com(S_{coord})$ gets a request j (event $GET(request(j, S, P))$) which destination is P, it stores information about request j in the $Outgoing$ table before sending the request. Each entry in $Outgoing$ stores information about an invocation made by S_{coord} on another object. This information is basically the same as for $Incoming$, except that the request itself is also stored[11] in the table (cf Section 3.2.2).

Whenever a reply to an outgoing request reaches the coordinator (event $RECEIVE(reply(j, S, P))$), $Com(S_{coord})$ checks whether this reply is duplicated. If so the reply is discarded, otherwise it is stored in $Outgoing$. The last event (event $ELECTION(Com(S_{coord}))$) corresponds to the first actions the new coordinator performs after the failure of the previous coordinator. Details are presented in Section 3.3.

3.2.2 Cohort

The cohort algorithm in Appendix B is basically the same as the coordinator algorithm, except that:

1. the cohort do not send outgoing requests and outgoing replies: it is the coordinator's exclusive duty;

2. outgoing requests are actually stored in $Outgoing$ for further retransmission (event $GET(request(i, S, P))$): the coordinator sends outgoing requests without storing them;

3. incoming requests may have $AWAITED$ as their status value: whenever the cohort receives a reply (event $RECEIVE(reply(i, S, P))$) *before* having received the corresponding request, the reply is stored in $Incoming$ with $AWAITED$ as its status value. For this reason, the cohort check whether an incoming request ($RECEIVE(request(i, C, S))$) is awaited. If so, the cohort set the status to $DONE$.

3.3 Fault tolerance

The cohort are actually backups of the coordinator. They stand silently and log incoming and outgoing invocations. Hence, failures of cohort members are harmless to the protocol. In contrast, the coordinator failure is critical: a new coordinator (chosen among the cohort) must take over to resume the protocol. Figure 5 shows execution diagrams describing how

[11] The coordinator *does not* store the request itself but the cohort do.

the protocol behaves in four situations involving coordinator failures:

a. no failures;

b. $Com(S_{coord})$ fails;

c. $Com(P_{coord})$ fails;

d. both fail.

Figure 5-a shows the normal case. $Com(C)$ sends request i to every $Com(S_k)$ which transmits the request to S_k. The latter generates request j which is sent by the coordinator[12] $Com(S_1)$ to every $Com(P_k)$. Request j is performed on each P_k and the reply is returned by the coordinator $Com(P_1)$ to every $Com(S_k)$. The latter transmits reply j to S_k which generates reply i. This reply is then sent to C by the coordinator $Com(S_1)$ via $Com(C)$.

Figure 5-b shows a case where the coordinator $Com(S_1)$ fails after having sent request j to every $Com(P_k)$. The cohort $Com(S_2)$ detects the coordinator failure and becomes the new coordinator[13]. The new coordinator retransmits (event $ELECTION(Com(S_{coord}))$ in Appendix A) every pending outgoing request (eg request j). The cohort $Com(P_2)$ discards request j because it has already received this request from the previous coordinator $Com(S_1)$. The coordinator $Com(P_1)$ checks whether reply j is available. If so, $Com(P_1)$ sends it to the surviving $Com(S_k)$. Otherwise, it discards request j to avoid a duplicated invocation (request j has already been transmitted to P_1 and the reply is not ready yet). Anyway, $Com(S_2)$ ends up with reply j and the invocation chain terminates normally when C receives reply i.

Figure 5-c shows a case where the coordinator $Com(P_1)$ fails after having received request j. The cohort $Com(P_2)$ detects the coordinator failure and becomes the new coordinator. According to the algorithm, $Com(P_2)$ retransmits every reply to incoming requests (eg reply j). Every $Com(S_k)$ receives reply j and the coordinator $Com(S_1)$ transmits reply i to $Com(C)$ to complete the invocation chain. Figure 5-d shows a case which is a combination of the two previous ones.

[12] Coordinators are shown by bold threads on Figure 5.

[13] An implementation of coordinator election is presented in Section 4.

4 Implementation of pre-filtering in GARF

The GARF system [Garbinato 94] is an object oriented environment aimed to simplify the programming of fault tolerant distributed applications. GARF provides libraries of fault tolerant objects which programmers use in their applications. This section presents the GARF object model and describes the support for active replication provided by GARF. This support includes an implementation of the invocation pre-filtering protocol described in the previous section.

4.1 The GARF object model

The GARF system assumes an object model in which every fault tolerant object is made of two distinct objects: a *data object* and a *behavioural object*. Data objects are application specific because they express application functionalities. They are programmed by the application programmer without taking care of both distribution and fault tolerance. Behavioural objects are reusable because they implement generic mechanisms related to both distribution and fault tolerance. They are provided by GARF libraries and used by application programmers.

A behavioural object intercepts all invocations sent or received by its associated data object. Figure 6 (a) shows the client/server invocation scheme as it is perceived by data objects (dashed arrow) and the invocation scheme as it is actually performed through behavioural objects (solid arrows). A behavioural object is actually implemented using two objects: an *encapsulator* and a *mailer*, as shown in Figure 6 (b). Each data object has a dedicated encapsulator and both are located on the same node. A copy of the mailer resides on every node where at least one *potential client* is located. An object is a potential client of a distributed object if it has the means to invoke it. Only encapsulators can be invoked across the network and the corresponding mailers alone can do it. The location of a data object is therefore transparent to its clients. The mailer can be seen as an *intelligent proxy* [Bijnens 94] of the server encapsulator for the client encapsulator. The mailer is cloned for each invocation and each time it is passed as argument accross the network.

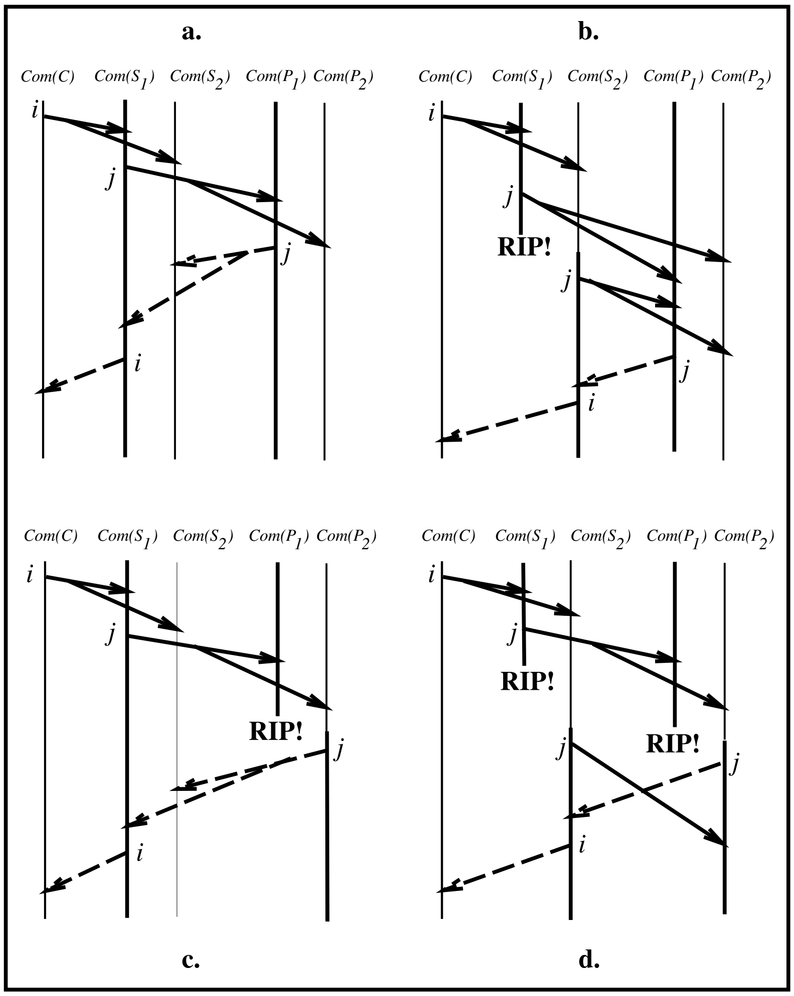

Figure 5: **Failure cases**

Figure 7 shows how an invocation of method **m** on server *S* by client *C*, is transformed by the GARF system into an invocation of **outRequest:to:** on *En(C)*, the client encapsulator. The latter receives the following arguments: the reified method #m and *Ma(S)*, a clone of server *S* mailer. 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 encapsulator, in order to forward #m to the server node. Eventually, *En(S)* invokes **m** on server *S*. Once method **m** has been performed by *S*, the result comes back to *C* following the reverse path, ie., 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 behaviour of *S* as server. All what **outRequest:to:** does defines the behaviour of *C* as client.

4.2 Active replication support in GARF

The behaviour of an *actively replicated* data object is built from an encapsulator of class ActiveReplica and a mailer of class Abcast. An ActiveReplica encapsulator is located on each node where a data object replica resides. It holds a reference on the local data object replica and belongs to an object group. On client nodes, Abcast mailers act as proxies of the encapsulator group. Mailers of class Abcast multicast invocations to the encapsulator group and ensure that every group member will receive the *same* sequence (ie total order) of invocations.

Figure 8 shows the same example as Figure 4. Given an object O, its communication object $Com(O)$ is implemented using O's encapsulator $En(O)$ and the mailers which $En(O)$ invokes. For instance, $Com(C)$ is implemented using $En(C)$ and $Ma(S)$; $Com(S_1)$ is implemented using $En(S_1)$

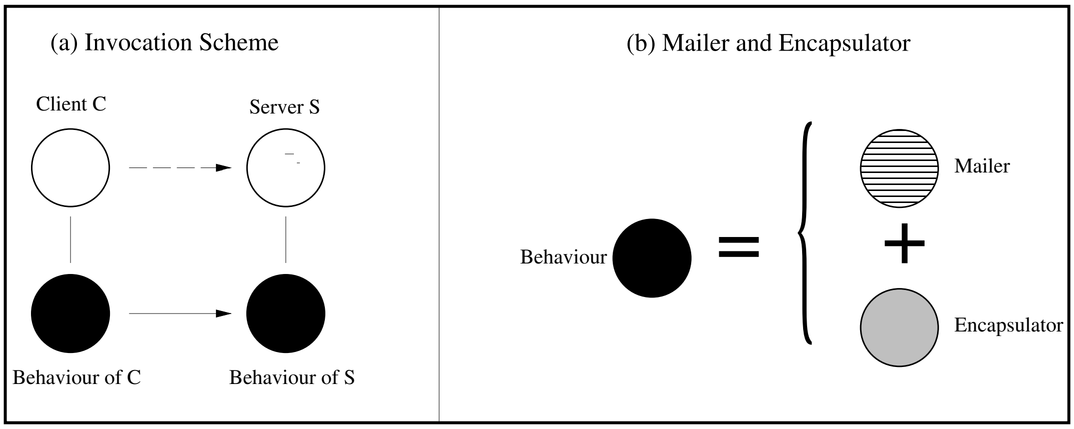

Figure 6: **Behaviours, mailers and encapsulators**

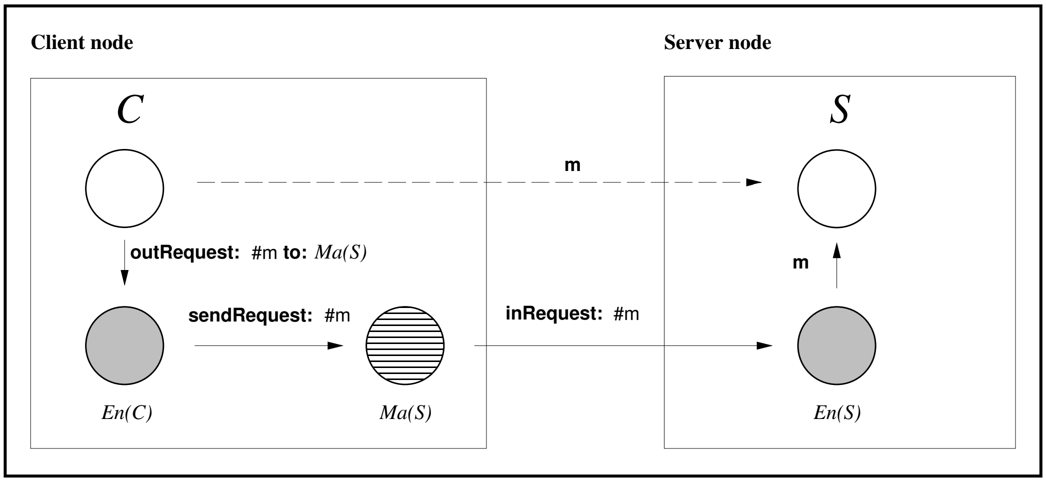

Figure 7: **Invocation path**

and $Ma(P)$. The encapsulator is the main part of a communication object. It actually performs the prefiltering protocol while the mailers are responsible for transmitting messages among the nodes.

Both classes ActiveReplica and Abcast are implemented in Smalltalk [Goldberg 83] on top of the *Dependable Object Toolkit* (*DOT*). DOT is a layer of the GARF system which implements replication management and group communications at the object level. It is built on top of the Isis toolkit [Birman 85, Birman 93] which provides support for both membership management and multicast communication of Unix process groups. The class Group provided by DOT, supports object group management and reliable multicasts of group invocations. Members of an object group are aware of the current group membership which is a list of object identifiers. This list is used to elect a new coordinator: for instance, each member chooses the first member of the list as the new coordinator. Class Group defines several multicast methods, among them **abcast:** which is used by instances of class Abcast.

Figure 9 presents the architecture of DOT based on Isis. A node is implemented using a pair of UNIX processes. The first process called *GARF/st*, contains the Smalltalk environment where objects reside. The second process called *DOT/c*, is written in C and linked with Isis libraries. Processes GARF/st and DOT/c communicate through Unix domain sockets. Isis manages groups of processes, while DOT deals with groups of objects. The mapping of DOT groups on Isis groups is straightforward. If a data object replica resides in the GARF/st process of some node and its encapsulator is member of a DOT group g, the DOT/c process on that node is

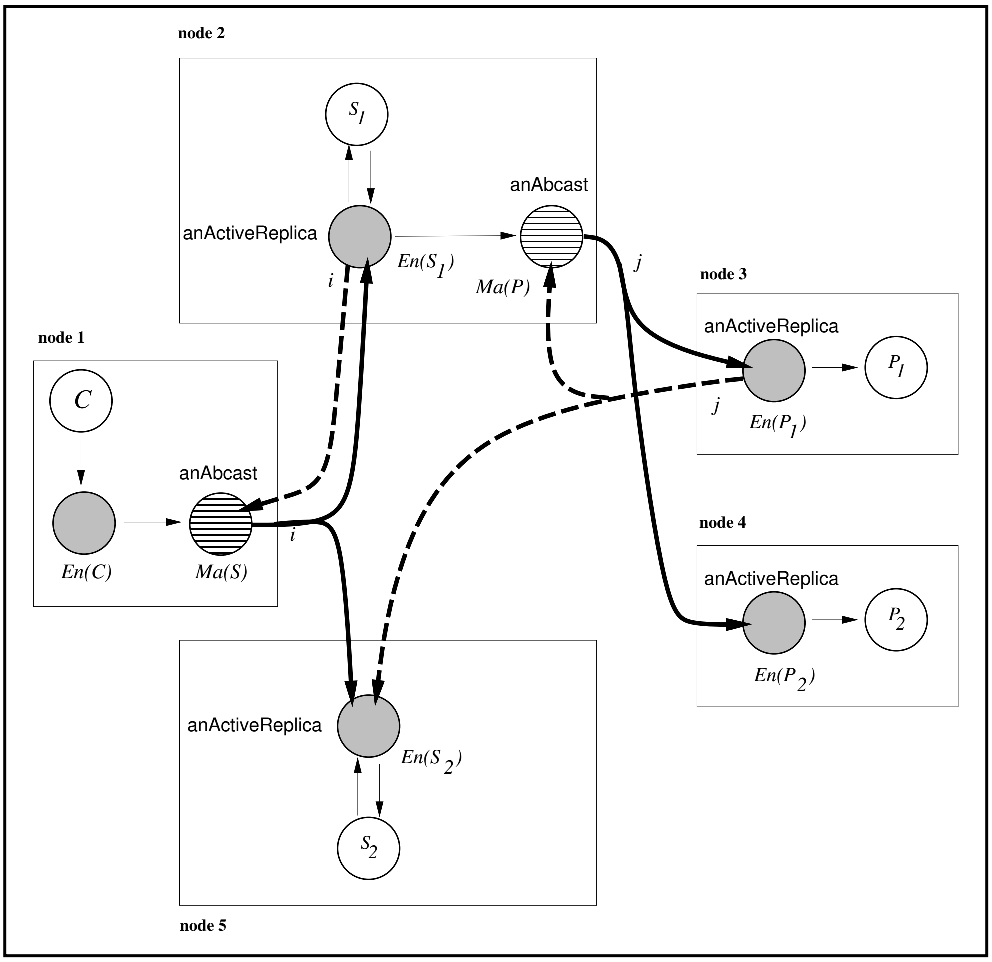

Figure 8: **Invocation pre-filtering in the GARF system**

member of an Isis group also named g. So a DOT/c process may be member of several groups. Instances of class Group store the group name g. When invoked by some multicast method, they pass it to the DOT/c process through a Unix domain socket. Process DOT/c then uses the equivalent Isis multicast primitive on that group.

5 Related work

Although the client-server model is widely used in distributed systems, its generalization to fault tolerance settings has been only partially investigated. Many authors [Cheriton 85, Mishra 89, Liang 90, Kaashoek 91, Cristian 91, Glade 92, Birman 93] and designers of distributed fault tolerant systems focus on server fault tolerance only and do not even mention client fault tolerance. Although it is not explicitly stated, these works assume that the roles of client and server are mutually exclusive: a software component is either a client or a server but cannot be both. This approach views a distributed system as a set of independent servers to which clients issue requests. The system state is confined in the servers and clients are stateless. Hence, fault tolerance is ensured only for servers. Servers are replicated but they are invoked only by non replicated clients and they do not need to invoke a third party server to handle a request. Replicated component interactions are thus avoided and the problem of duplicated interactions no more exists!

However, a few authors [Cooper 86, Powell 91, Elnozahy 92, Wood 93] do not share this view of distributed systems and provide mechanisms to support replicated component interactions. The designers of Circus [Cooper 86] felt early (1984) the need for such support. They propose a *replicated procedure call* facility which allows a *troupe* (i.e a set of replicated software components) to invoke an-

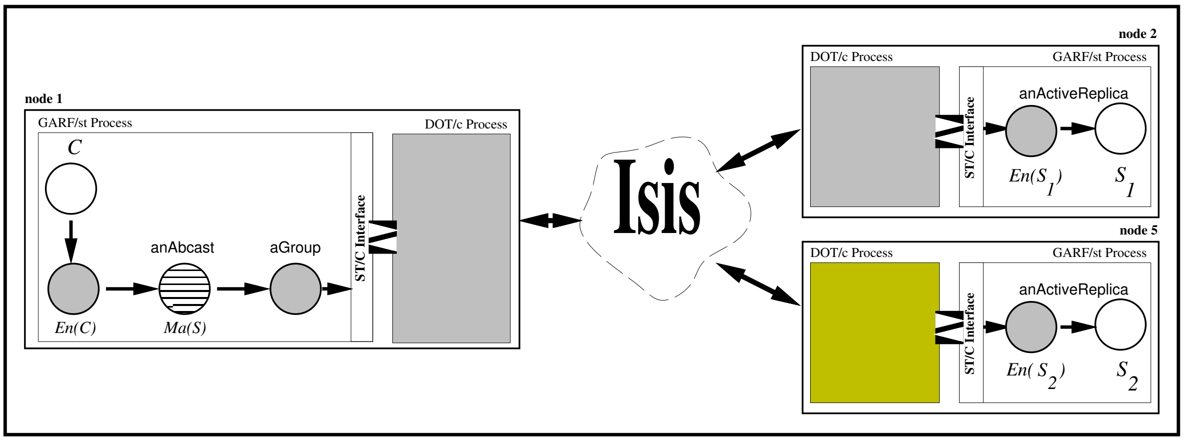

Figure 9: **Architecture of DOT based on Isis**

other troupe. Each troupe is actively replicated and specific components called *collators* perform post-filtering to avoid duplicated invocations. Replicated RPCs of Amoeba [Wood 93] implement a sort of pre-filtering to avoid duplicated interactions. The algorithm is based on the coordinator-cohort mechanism provided by ISIS [Birman 85]. Although this mechanism is not defined for replicated objects, it resembles the protocol presented in section 3 but it enforces a tighter coordination between the replicas.

Objects are particularly well suited to design distributed systems because they implement naturally the client-server model. An object is alternately a client (whenever it invokes) and a server (whenever it is invoked). Thus, it seems reasonable to have support for replicated object interactions within object oriented distributed systems concerned with fault tolerance. Several systems [Shrivastava 91, Gourhant 92] provide support for replicated objects but only a few authors mention what happens when replicated objects communicate. The SOR project [Gourhant 92, Dickman 92] proposes the *fragmented object* model in order to implement replicated objects. A fragmented object is capable of implementing several replication policies. However, it is not clear how several fragmented objects interact and how duplicated interactions are avoided. In Arjuna [Little 91, Little 93], fault tolerant objects are actively replicated. The authors mention briefly a filtering mechanism (post-filtering at the RPC level) to discard duplicated invocations. The designers of the ANSA model [Olsen 91] propose a post-filtering approach in order to discard duplicated invocations of actively replicated objects. Their approach is based on *interface groups*. A replicated object is implemented using an interface group which provides two specific objects called *distributor* and *collator*. The distributor sends each invocation of the interface group to every replica. Its plays the same role as the mailer in the GARF model (see Section 4.1). The collator actually performs invocation post-filtering and discard duplicated invocations.

6 Conclusion

This paper shows that interactions of replicated objects are not straightforward. Whenever actively replicated objects communicate, they are faced with the duplicated invocation problem. The latter can be solved thanks to a pre-filtering mechanism which is currently implemented in the GARF system. However, duplicated invocations is only one of the numerous problems which are raised whenever replicated objects interact.

Another problem is the conflict between the notion of replication policy and the invocation paradigm. The replication policy of an object defines how the replicas of *this* object receive, perform and reply to invocation requests. In contrast, the invocation paradigm defines a client-server interaction model which is *independent* of object characteristics. According to this model, an invocation consists in sending a single request to an object and getting back a single reply from the object.

Assume that a client object C invokes an actively replicated server object S. The client C must send its invocation request to *every* S replica. Since each replica of S replies to C, the latter must cope with *several* replies to its single invocation. Hence, C must be aware of the replication policy of S in order to communicate with the server. This example shows that the invocation model is no more respected when replication is considered. In addition, it shows that the replication policy is a server's characteristic which must be known by the client.

A reasonable solution to the conflict depicted above is an approach where replication policies are part of the object invocation service. In this context, whenever object C invokes object S, the invocation service treats both the request and the reply according to the replication policies of both objects. Each replication policy is known only by the replicated object itself and by the invocation service. From an external point of view, each object seems to encapsulate its own replication policy. Furthermore, a replication policy can be naturally integrated into the invocation service because the latter, as the former, implement a communication path between an object and its clients.

References

[Bijnens 94] S. Bijnens, W. Joosen & P. Verbaeten. *A Reflective Invocation Scheme to Realise Advanced Object Management*. In R. Guerraoui, O. Nierstrasz & M. Riveill, editeurs, Object Based Distributed Programming. Springer Verlag, 1994.

[Birman 85] K.P. Birman. *Replication and Fault-Tolerance in the ISIS system*. In Proceedings of the 10th ACM Symposium on Operating Systems Principles, pages 79–86, December 1985.

[Birman 93] K.P. Birman. *The Process Group Approach to Reliable Distributed Computing*. Communications of the ACM, vol. 36, no. 12, pages 36–53, december 1993.

[Cheriton 85] D.R. Cheriton & W. Zwaenepoel. *Distributed Process Groups in the V kernel*. ACM Transactions on Computer Systems, vol. 3, no. 2, pages 77–107, May 1985.

[Cooper 86] E.C. Cooper. *Replicated Procedure Call*. ACM Operating Systems Review, vol. 20, no. 1, pages 44–56, January 1986.

[Cristian 91] F. Cristian. *Understanding Fault-Tolerant Distributed Systems*. Communications of the ACM, vol. 34, no. 2, pages 56–78, February 1991.

[Dickman 92] P. Dickman & M. Makpangou. *Refinement of the Fragmented Object Model*. In Proceedings of the International Workshop on Object orientation in Operating Systems, pages 230–234. IEEE, september 1992.

[Elnozahy 92] E.N. Elnozahy & W. Zwaenepoel. *Replicated Distributed Processes in Manetho*. In Proceedings of the 22nd International Symposium on Fault-Tolerant Computing, July 1992.

[Garbinato 93] B. Garbinato, R. Guerraoui & K.R. Mazouni. *Programming Fault-Tolerant Applications Using Two Orthogonal Object Levels*. In Proceedings of the 8th International Symposium on Computer and Information Sciences (ISCIS-8), 1993.

[Garbinato 94] B. Garbinato, R. Guerraoui & K.R. Mazouni. *Distributed Programming in GARF*. In R. Guerraoui, O. Nierstrasz & M. Riveill, editeurs, Object Based Distributed Programming, numéro 901 in Lecture Notes in Computer Science. Springer Verlag, 1994.

[Glade 92] B.B. Glade, K.P. Birman, R.C.B. Cooper & R. van Renesse. *Light-Weight Process Groups*. In Proceedings of the OpenForum 92

Technical Conference, November 1992.

[Goldberg 83] A.J. Goldberg & A.D. Robson. Smalltalk-80: The language and its implementation. Addison Wesley, 1983.

[Gourhant 92] 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.

[Kaashoek 91] F. Kaashoek & A. Tanenbaum. *Group Communication in the Amoeba Distributed Operating System*. In Proceedings of the 11th International Conference on Distributed Computing Systems, pages 222–230, May 1991.

[Liang 90] L. Liang, S.T. Chanson & G.W. Neufeld. *Process Groups and Group Communications: Classifications and Requirements*. IEEE Computer, vol. 23, no. 2, pages 56–65, February 1990.

[Little 91] M.C. Little. *Object Replication in a Distributed System*. PhD thesis, University of Newcastle-upon-Tyne (UK), September 1991.

[Little 93] M.C. Little, D.L. McCue & S.K. Shrivastava. *Maintaining Information about Persistent Replicated Objects in a Distributed System*. In Proceedings of the 13th International Conference on Distributed Computing Systems, pages 491–498. IEEE Computer Society Press, May 1993.

[Mazouni 94] K.R. Mazouni, B. Garbinato & R. Guerraoui. *Programmation d'une application distribuée avec l'environnement GARF*. In Proceedings of the 3rd Maghrebian Conference on Software Engineering and Artificial Intelligence. Maghrebian Information Processing Society, 1994.

[Mishra 89] S. Mishra, L.L Peterson & R.D. Schlichting. *Implementing Fault-Tolerant Replicated Objects Using Psynch*. In Proceedings of the 8th Symposium on Reliable Distributed Systems, pages 42–52. IEEE, October 1989.

[Olsen 91] M.H. Olsen, E. Oskiewicz & J.P. Warne. *A Model for Interface Groups*. In Proceedings of the 10th Symposium on Reliable Distributed Systems, pages 98–107, October 1991.

[Powell 91] D. Powell, editeur. Delta4: A generic architecture for dependable computing. Springer-Verlag, 1991.

[Schneider 90] F.B. Schneider. *Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial*. ACM Computing Surveys, vol. 22, no. 4, pages 299–319, December 1990.

[Shrivastava 91] S.K. Shrivastava, G.N. Dixon & G.D. Parrington. *An Overview of the Arjuna Distributed Programming System*. IEEE Software, 1991.

[Wood 93] M. Wood. *Replicated RPC Using Amoeba Closed Group Communication*. In Proceedings of the 13th International Conference on Distributed Computing Systems, May 1993.

Appendix A: algorithm of the coordinator

if $event = RECEIVE(request(i, C, S))$:	Request i has been received from C.
$\rightarrow x := FIND(request(i, C, S), Incoming);$	
if $x <> NIL$	
then **if** $Incoming[x].status = DONE$	
then $SEND(Incoming[x].reply);$	The reply to request i is retransmitted.
else $DISCARD(request(i, C, S));$	Request i is duplicated and discarded.
fi	
else $x := NEW(Incoming);$	A new entry is allocated in $Incoming$.
$Incoming[x].id := (i, C, S);$	
$Incoming[x].status := PENDING;$	
$DELIVER(request(i, C, S));$	Request i is delivered to S_{coord}.
fi	
$\Box$ $event = GET(reply(i, C, S))$:	A reply to request i has been generated.
$\rightarrow x := FIND(request(i, C, S), Incoming);$	
$Incoming[x].reply := reply(i, C, S);$	The reply is stored in $Incoming$ for further retransmission
$Incoming[x].status := DONE;$	
$SEND((reply(i, C, S));$	and sent to C.
$\Box$ $event = GET(request(j, S, P))$:	Request j from S to P has been generated.
$\rightarrow x := NEW(Outgoing);$	
$Outgoing[x].id := (j, S, P);$	
$Outgoing[x].status := PENDING;$	
$SEND(request(j, S, P));$	Request j is sent to P.
$\Box$ $event = RECEIVE(reply(j, S, P))$:	A reply to request j has been received from P.
$\rightarrow x := FIND(request(j, S, P), Outgoing);$	
if $x <> NIL$	
then **if** $Outgoing[x].status = DONE$	
then $DISCARD(reply(j, S, P));$	The reply is duplicated and discarded.
else $Outgoing[x].reply := reply(j, S, P);$	The reply is stored in $Outgoing$
$Outgoing[x].status := DONE;$	for further retransmission
$DELIVER(reply(j, S, P));$	and delivered to S_{coord}.
fi	
fi	
$\Box$ $event = ELECTION(Com(S_{coord}))$:	$Com(S_{coord})$ is the new coordinator.
$\rightarrow$ **do** $\forall x \mid Outgoing[x].status = PENDING$	
$SEND(Outgoing[x].request);$	Pending outgoing requests are retransmitted.
od	
do $\forall x \mid Incoming[x].status = DONE$	
$SEND(Incoming[x].reply);$	Replies to incoming requests are retransmitted.
od	
fi	

Appendix B: algorithm of the cohort

```
if event = RECEIVE(request(i, C, S)) :                         // Request i has been received from C.
  → x := FIND(request(i, C, S), Incoming);
    if x <> NIL
      then if Incoming[x].status = AWAITED
             then Incoming[x].status := DONE                    // Request i has been received after the reply!
             else DISCARD(request(i, C, S));                    // Request i is duplicated and discarded.
           fi
      else x := NEW(Incoming);
           Incoming[x].id := (i, C, S);
           Incoming[x].status := PENDING;
           DELIVER(request(i, C, S));                           // Request i is delivered to S_cohort.
    fi

□ event = GET(reply(i, C, S)) :                                // A reply to request i has been generated.
  → Incoming[x].reply := reply(i, C, S);                        // The reply is stored in Incoming for further retransmission
    Incoming[x].status := DONE;                                 // BUT the cohort DO NOT send replies.

□ event = GET(request(j, S, P)) :                              // Request j from S to P has been generated.
  → x := NEW(Outgoing);
    Outgoing[x].request := request(j, S, P);                    // The request is stored in Outgoing for further retransmission
    Outgoing[x].id := (j, S, P);
    Outgoing[x].status := PENDING;                              // BUT The cohort DO NOT send requests.

□ event = RECEIVE(reply(j, S, P)) :                            // A reply to request j has been received from P.
  → x := FIND(request(j, S, P), Outgoing);
    if x <> NIL
      then if Outgoing[x].status = DONE
             then DISCARD(reply(j, C, S));                      // The reply is duplicated and discarded.
             else Outgoing[x].reply := reply(j, S, P);          // The reply is stored in Outgoing
                  Outgoing[x].status := DONE;                   // for further retransmission
                  DELIVER(reply(j, S, P));                      // and delivered to S_cohort.
           fi
      else x := NEW(Outgoing);                                  // The reply reached the cohort before the request!
           Outgoing[x].id := (j, S, P);
           Outgoing[x].status := AWAITED;
    fi
fi
```