# Nesting Actions through Asynchronous Message Passing : the ACS Protocol

**Rachid Guerraoui** **Riccardo Capobianchi**
**Agnes Lanusse** **Pierre Roux**

**CE Saclay DEIN/SIR**
**91191 Gif sur Yvette Cedex France**
**E_mail : (guerraoui,capobianchi,lanusse,roux)@leti.cea.fr**
**Phone :(+33)(+1) 69.08.60.53**
**Fax : (+33)(+1) 69.08.83.95**

**Abstract.** This paper describes an original object communication protocol, named ACS (Apply, Call, Send), that we adopted in KAROS (Kernel of an Action-based Reliable Object System) to deal with concurrency control and failure recovery in distributed applications. The basic idea in ACS is to associate each request message to an atomic action. ACS allows to build atomic action trees representing the logical nesting of services. Moreover, it gives to the programmer full control on the granularity of atomicity and provides a fairly natural model for grouping objects inside actions to ensure the system's global consistency. By using ACS, objects may be seen as reliable elements that can be composed to build up reliable distributed applications.

## 1 Introduction

The intrinsic properties of object orientation address the evolutionary aspect of complex programs and propose a methodology for designing programs as sets of communicating modules [17]. On the other hand, distributed systems often lack programming methodologies and high-level languages that can hide low-level primitives for inter-process communication and synchronization, resource management and access to remote information. The merging of distributed systems and object-based programming has given birth to an important investigation area, whose aim is to build object-based distributed systems well suited for distributed applications programming.
The consistency of such applications is based on assertions that are made by objects about the state of other objects. Nevertheless, since objects may execute concurrently at different nodes, executions interferences or failures may modify those assertions and thus corrupt the overall system consistency.
A well known solution to this problem is to enclose related objects executions inside atomic actions resembling transactions in databases [1]. Atomicity as in the sense of Weihl [27] implies serialisability, failure atomicity and permanence of effects. The serialisability property means that concurrent actions are isolated from each other. Failure atomicity combined with permanence of effects implies that each action is either executed completely and its effects are permanent or it is aborted and its effects are discarded.

Considering actions as monolithic programs like in classical databases introduces two major drawbacks in a distributed environment:

- in an action-based system, concurrency control is provided for inter-action concurrency. However, as there is no concurrency control within an action, programmers have either to handle the synchronization of parallel threads inside the action boundaries or to forbid intra-action concurrency, with the risk of heavily impacting on the system efficiency.

- if any local failure occurs during an action execution, this one is aborted and its effects on all the objects related by the action are discarded. Local failures can not be masked to take advantage of partial system availability. A long running distributed action whose execution time comes near the MTBF (Mean Time Between Failures) of less reliable components would probably never complete and objects' modifications would never occur.

A powerful extension of the classical action scheme is the *nested action* model proposed by Moss [18], that permits both to implicitly control intra-action concurrency and to reduce failures effects into local parts of a computation. In this model, an action may be broken into pieces. Each piece is a subaction which in its turn may be broken into more subactions. The overall action is thus seen as a tree of nested actions at different levels.
Two subactions may safely be executed concurrently because they are considered as different, according to the underlying concurrency control. Moreover, subactions can commit and abort independently of each other and an action can abort without forcing its parent to abort. This is very useful when an action can commit even though some of its subactions have failed. Both failure atomicity and serialisability are guaranteed for subactions. However, a subaction commits only when its parent action commits, so that the permanence of effects is guaranteed only for top-level actions.

This paper presents an object communication protocol, named ACS (*Apply*, *Call*, *Send*), that merges the nested actions model with the model of nested asynchronous request messages. The protocol is used for concurrent objects communication in KAROS: an exploratory language designed for reliable distributed applications.
The paper is organized as follows. Section 2 offers a reminder of previous related works and shows how those approaches either complicate the programmer's job or produce unwieldy inefficiencies. Section 3 discusses design considerations related to asynchronous frameworks. Section 4 presents the ACS protocol through its use in the KAROS language. Section 5 gives a simple example of a KAROS application focusing on the communication protocol. Section 6 concludes with some final remarks and presents some related open research problems.

## 2 Previous approaches

Many object-based systems, either distributed or designed with distribution in mind, have provided some nested actions facilities (at least for serialisability). We have classified those approaches into two classes.
The first one, which we called the *explicit approach*, contains languages giving to the programmer specific constructs to design actions as internal parts of methods executions. The second one, called the *implicit approach*, contains languages that merge actions and services. We call service the task consisting of sending a message to its target (a server) and executing the corresponding method. In a communication requesting a reply message, the service includes as well the invoice of the reply from the server to the client.

### 2.1 The explicit approach

Some languages, like Eden [21], Hybrid [20], Meld [12] and Arjuna [7] have proposed linguistic constructs which can be used explicitly by the programmer to identify logical units of computation as actions.
Meld and Hybrid have proposed an *action block* construct ensuring serialisability. In Eden's *resource manager* and in Arjuna, actions are represented by first class objects used as any other object in the system.

**Action blocks.** An *action block* is a set of statements bound by two linguistic constructs and enclosed inside a method execution. Each statement involves either an internal statement or a message passing operation. The action starts with the first statement and terminates when it reaches the last statement and when all the cascading statements (i.e. the statements involved by message passing) are terminated.

***Example :*** Meld provides actions facilities through the use of angle brackets. Every time a programmer wants to identify a set of statements as an atomic activity, he places it inside angle brackets:

```
CLASS F ::= d:E;   e:integer;
   METHODS:
      R() -->
      < e := d.P(1,2);>
   END CLASS
```

The statement " *e := d.P(1,2)* " is executed as an atomic activity within each execution of the method *R()*.

Actions can be arbitrarily nested by directly nesting angle brackets or via invocations of methods containing angle brackets.

**Actions as first class objects.** Actions are represented by first-class objects addressed by other objects through operations like *begin*, *commit* and *abort*.

***Example :*** The Eden resource manager system provides a built-in type *TransactionManager* which supports three operations: *BeginTransaction*, *AbortTransaction* and *CommitTransaction*. Every time a user wants to identify an action, he creates an instance of a *TransactionManager*. The transactions are explicitly nested: the *BeginTransaction* operation takes as parameter a parent action name and creates a subaction of the parent action:

```
PROCEDURE  proc(ParentTM,..)
      newTM.BeginTransaction(ParentTM,..)
         ....    body    ....
      newTM.CommitTransaction(..)
END  proc
```

**Discussion.** Even though the explicit approach gives to the programmer full control over the actions boundaries, it complicates its work as well as the readability of its programs. In fact, the programmer has to deal with two logical trees: the tree representing the nested actions and the tree representing the nested services, both representing nested logical units of computations. The programmer must break the computations into pieces

representing logical units of execution and has to write explicit code for each piece. A faulty code would lead to inconsistent system executions.

### 2.2 The implicit approach

An intuitive solution is to merge the nesting of actions and the nesting of methods executions since both represent logical computations. Each request message issued from a client is enclosed inside an atomic activity whose atomicity is assured by the underlying system. Every object involved by the service is considered as affected by the corresponding action. If the service succeeds then the action succeeds, otherwise the service fails and the corresponding action is aborted discarding its effects on the affected objects. When a service terminates successfully its effects are permanent and thus are not undone in case of failure, e.g. if a server node crashes. This gives to the message passing metaphor a powerful semantic for reliable distributed computing.

There have been some propositions [15] in this direction, all based on the *synchronous Remote Procedure Call* mechanism (RPC) [2].
When a client asks for a service through a RPC, an action is created and associated to the service. If the service involves another RPC to accomplish a subservice, a subaction of the original action is created and associated to the subserver. In case of a service failure, the client is not forced to abort. For example, the client may choose to ask for another service or to ignore the subaction failure.

> ***Example :*** An Argus [16] program consists of a set of *Guardians* (large grain objects) executing on different nodes of a distributed system. Each *Guardian* addresses another *Guardian* by calling one of its *handlers* (methods) through a RPC. The server *Guardian* creates an internal thread to execute each of the incoming invocations.
> When a *handler* is invoked, a new subaction is created. The subaction encloses the sending of the message, the execution of the *handler* and the reply message. If there is any system failure, the system replies with a failure exception. The invoked *handler* can also abort the subaction and terminate in a user-defined exception. The client may execute an alternative code if the programmer has expected the exception (caused by a failure or user-defined):

```
guard.hand(arg)
      %  the "hand" handler is invoked on the "guard" Guardian
except when failure()
      %  alternative code in case of a service failure
end
      %  normal code in case of a successful service
```

The *atomic RPC* model provides a powerful way of designing atomic services. However, the systematic approach of nesting does not always provide the best choice. Before committing, the parent action should wait for the nested action outcome, and if the parent action aborts then the nested action will be also forced to abort. In fact, there are many cases in which a subservice does not need to lie within the original action boundaries. A RPC involving a benevolent side effect, such as a garbage collection or a statistics reporting, does not need to be designed as a subaction but as an independent top-level service. The garbage collection for example does not need to abort if the parent action fails, which in its turn does not need to wait for the garbage collection outcome.

Argus proposes a solution to this issue by providing a linguistic construct that allows to create top-level actions inside a handler:

```
hand = handler()
       .....
       enter topaction
       .... body ....
       end
end hand
```

This solution presents a drawback: since there is no uniform syntax for action and subaction creation, it is not possible for the programmer of a *Guardian* client to compose *Guardian* servers' top-level actions for building larger actions; this impacts on software composition of reliable components.

## 3 Atomic asynchronous remote invocations: design issues

RPC is a well understood abstraction (the procedure call) allowing easy support of type checking; it is now a widely used interprocess communication facility in distributed systems [6, 22]. Nevertheless, the client of a RPC is always blocked during the remote execution: this is not always the desired behaviour. Indeed, in many cases the client must be able to turn its attention to another job to take advantage of the available parallelism. An asynchronous mode of message passing implicitly allows a client to continue executing while the remote procedure is taking place. This is very useful if the client does not need a reply, or when a client sending multiple requests to different servers needs only to wait for the first reply. Walker [26] has shown how asynchronous objects invocations can provide such facilities while keeping the benefits of RPCs.

ABCL/1 [29], Concurrent Smalltalk [28], Orient [14], Hybrid [20] and PO [5] give the programmer the possibility to use *asynchronous message passing* as well as *synchronous remote procedure call* (RPC). ACT++ [11] and Concurrent-Eiffel [4] propose a systematic asynchronous communication protocol. Unfortunately, none of those languages have provided *implicit atomic service* semantics. If the programmer wants to design a service as an atomic activity he has to write an explicit, and perhaps faulty code for each service. We believe that both *asynchronous message passing* and *atomic service* semantics are important enough to consider for merging as implicit linguistic constructs in a programming language addressing reliable distributed applications.

If we try to improve the RPC model by moving to an asynchronous framework we are faced with three main issues:

- in an *asynchronous message passing* system, the parent action will not block when a message is issued; if the client does not provide explicit concurrency control, the action and its subaction could execute concurrently and corrupt the accessed objects consistency, because an action and its subaction are not considered as two different actions. Another subaction is thus necessary to continue the client execution.

- if a RPC service fails, the client's control flow is implicitly affected by an exception mechanism (e.g Argus). This is not so easy to accomplish in an *asynchronous message passing* system since the client is actually carrying on its execution.

- as we already pointed out in the previous subsection, the systematic approach of nesting does not always provide the best choice. When a subservice is designed as a nested action, the client and the server have to mutually wait for each other before committing the service and subservice execution effects, which may not always be logically related.

We have designed a communication protocol, named ACS, that provides *implicit atomic service* semantics while avoiding the above mentioned drawbacks.

# 4 The ACS protocol

ACS has been designed and implemented within an exploratory concurrent object-oriented language named KAROS [8] whose design attempts to address distributed applications. Hereby we will present an overview of KAROS focusing on the aspects related to the communication protocol.

## 4.1 An overview of KAROS

KAROS provides two kind of objects: *ActiveObjects* and *DataObjects*. *ActiveObjects* are logical units of distribution; they are uniquely referenced and their references may be known and passed to other *ActiveObjects* in the system. Each *ActiveObject* state consists of a set of references to other *ActiveObjects* and of local *DataObjects*.
A *DataObject* is always local to an *ActiveObject*; it is created inside a parent *ActiveObject* and its reference may not be passed to other *ActiveObjects*. When executing a method, the *ActiveObject* may communicate with local *DataObjects* as well as with reachable *ActiveObjects*. The method may terminate either successfully (possibly replying to a request) or by an explicit failure, aborting the current action:

```
Method(class,meth)
{ ..............                                    /*  method body  */
   Reply(result) ;
     /*  the method terminates successfully and returns a result */
}

Method(class,meth)
{ ..............                                    /*  method body  */
   Fail();
                          /*  the method terminates by failing  */
}
```

*ActiveObjects* communicate by passing data values and references in an asynchronous way: after sending a message, an *ActiveObject* can immediately carry on its execution.
KAROS provides the use of normal variables as *implicit futures* [4, 13]. When a client expects a reply, it is not forced to wait until it tries to retrieve it by using the value of the *future variable*. However, the client may ask if the result has arrived (*Awaited* construct) or decide to wait for it anyhow (*Wait* construct):

```
res = Call(server, class, method) << Arg1 << ... ;
                                /* asynchronous message passing;
                                res is an implicit future variable */
.....
Bool arrived = Awaited(res) ;
                                /* testing for the result arrival */
.....
Wait(res) ;                     /* waiting for the result arrival */
```

We have extended the *future reply* mechanism so that a client of an asynchronous service may also ask if the service has failed or not:

```
Bool arrived = Failure(res) ;          /* testing for a service failure */
```

A failure may occur during the client execution, in the communication system or during the server execution.

Since KAROS is an exploratory language, we have chosen to implement it as a C++ [24] class hierarchy on a Sun workstation under Unix BSD 4.0, along the line of ACT++ [11]. KAROS makes extensive use of C++ operators' overloading and macros. A distributed version is under implementation using a process allocation mechanism described in [3].

### 4.2 The communication protocol

*ActiveObjects* communicate through three kind of *asynchronous message passing* : *Apply*, *Call* and *Send*, constituting the *ACS* protocol.

**Apply.** When a client issues an *Apply* message, two concurrent subactions are created (fig. 1). The first subaction goes through the server and encapsulates the service; it terminates when the result arrives to the client. The second subaction encapsulates the client execution that starts just after the *Apply* message statement. The two subactions are safely performed concurrently since they are considered as different actions. If a failure occurs in any of the subactions, the system aborts the parent action (fig. 2).

```
res = Apply(server, class, method) << Arg1 ... << ArgN ;
```

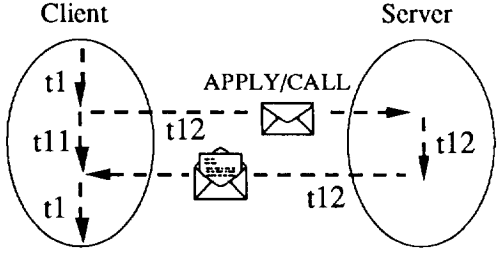


Fig. 1

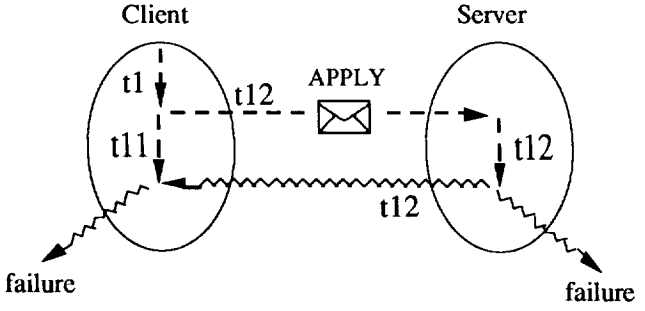


Fig. 2

**Call.** As in the *Apply* model of request message, two similar subactions are created (fig. 1). However, if any failure occurs in the communication system or during the server execution, the parent action is not forced to abort (fig. 3). The client may know that the request has failed, and it may consequently choose to abort or to continue its action. Different subactions are thus allowed to fail independently of each other, even though they are related by the same global action. This is very useful when a service may be considered as correct although some of its subservices have not been accomplished.

```
res = Call(server1, class, method) << Arg1  ...  << ArgN ;
if ( Failure (res) )
     res = Call(server2, class, method) << Arg1  ...  << ArgN ;
                                          /* alternative code */
else  .....                               /*  normal code  */
```

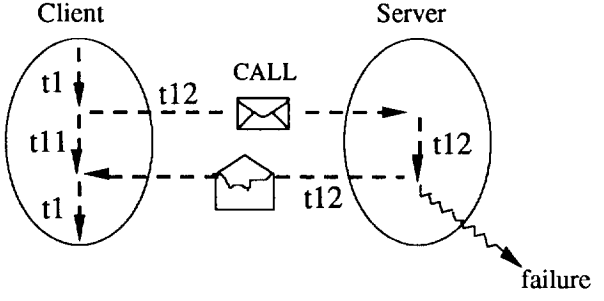


Fig. 3

**Send.** After issuing a *Send* message, the client continues executing inside its current action (fig. 4). It does not rely on the server execution and does not expect any reply from the server. The system creates a top-level independent action to enclose the message passing and the remote invocation. This kind of message passing provides a simple way for safely breaking atomicity when an independent activity has to be triggered.

```
Send(server, class, method) << Arg1  ...  << ArgN ;
```

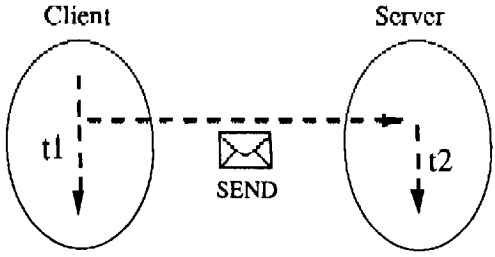


Fig. 4

### 4.3 Action management

When an *ActiveObject* receives a message it creates an action manager associated to the method execution that keeps track of the action history (fig. 5). The action manager records all the local and external created subactions as well as the local *DataObjects* and *ActiveObjects* references accessed by the action (and its subactions).

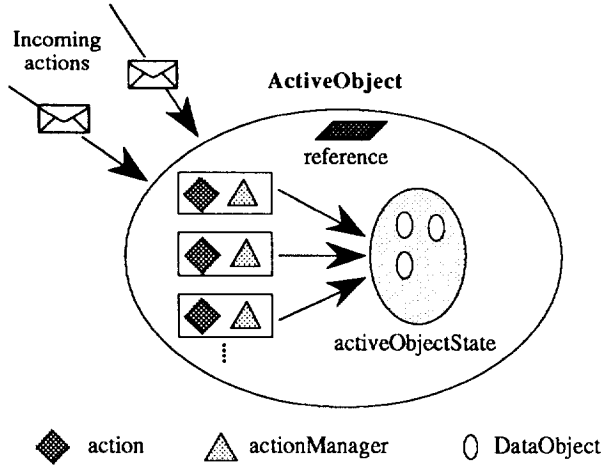


Fig. 5

The action terminates when the corresponding method execution and all its nested subactions terminate.
If the action is a subaction, the *ActiveObject* builds the reply message and sends it back to the client *ActiveObject*: the subaction action commits when the *ActiveObject* receives a reception acknowledgment. If the action is a top-level one, the action manager starts the execution of a commit protocol. To guarantee that every modification to accessed objects is either permanent or is not done at all, we have considered a two phase commit protocol [10].

Failure atomicity and serialisability are provided for *DataObjects*, which are designed as atomic objects [27]. Atomic objects provide the appropriate synchronization and recovery

needed, in such a way that the actions using them appear to be atomic. Global atomicity is guaranteed if all the *DataObjects* local to an *ActiveObject* are atomic.
A KAROS atomic object is a *DataObject* that inherits directly or indirectly from a specific class *AtomicObject*. In the first implementation of the *AtomicObject* class, we have chosen a dynamic protocol for concurrency control [27] similar to the nested two phase locking algorithm [19]. We did not consider semantics concurrency control [23]; in fact, an action conflicts with every action which is not one of its ancestors. Failure atomicity is ensured by a *backup algorithm* [19]. However, KAROS proposes a set of basic classes that may be refined through inheritance to provide a wide range of concurrency control protocols [10].

## 5 A simple example

We will try to show the usefulness of ACS using its implementation in the KAROS language by the way of a simple example, concerning the organization of a university-sponsored conference (summer school style).
We suppose that the conference committee agrees on a subject, a date and a venue; the task of organizing a conference consists in gathering a sufficient number of concerned professors/speakers, in reserving a hall in the venue for the chosen date, in booking an appropriate number of hotel rooms for the speakers, and finally sending the invitations to a selected audience. We argue as well that a professor whose agenda is free for the conference day cannot refuse the invitation, and that all professors have to be booked in the same hotel for the sake of conviviality.

Our conference organizing system is naturally distributed, and is made up of *ActiveObjects* residing on different nodes and exchanging messages.
The main *ActiveObjects* are:

- the Conference Manager (cm), accepting requests from the conference committee;
- the Professors Manager (pm), sending requests to the Professor Agendas;
- the Venue Manager (vm), sending requests to the Hall Agendas;
- the Hotels Manager (hm), sending requests to the Hotel Booking Systems;
- the Audience Manager (am), sending out invitations and other information to the concerned people Mailing Systems.

The organization of a conference is carried out inside a single action (created by the user requesting a service at the terminal), that generates several levels of nested actions (fig. 6). The user request is translated into a *Call* message to the *cm ActiveObject*, containing the user-specified parameters (subject, site and date of the conference):

```
Bool rep = Call (cm, ConferenceManager, organize_conference) <<
                                        subject << venue << date ;
```

This message creates a concurrent subaction in the *cm*, which in its turn creates other concurrent subactions by sending messages (with the relevant parameters) to the *pm*, *vm*, *hm* and *am ActiveObjects*.
Those are *Apply* messages because if any failure occurs during one of the servers' executions also the *cm* execution has to fail (the conference cannot be organized if one of those *ActiveObjects* doesn't perform its task). It can be noted that the *pm* returns a reply that is sent as a parameter to the *hm* (number of participant professors).

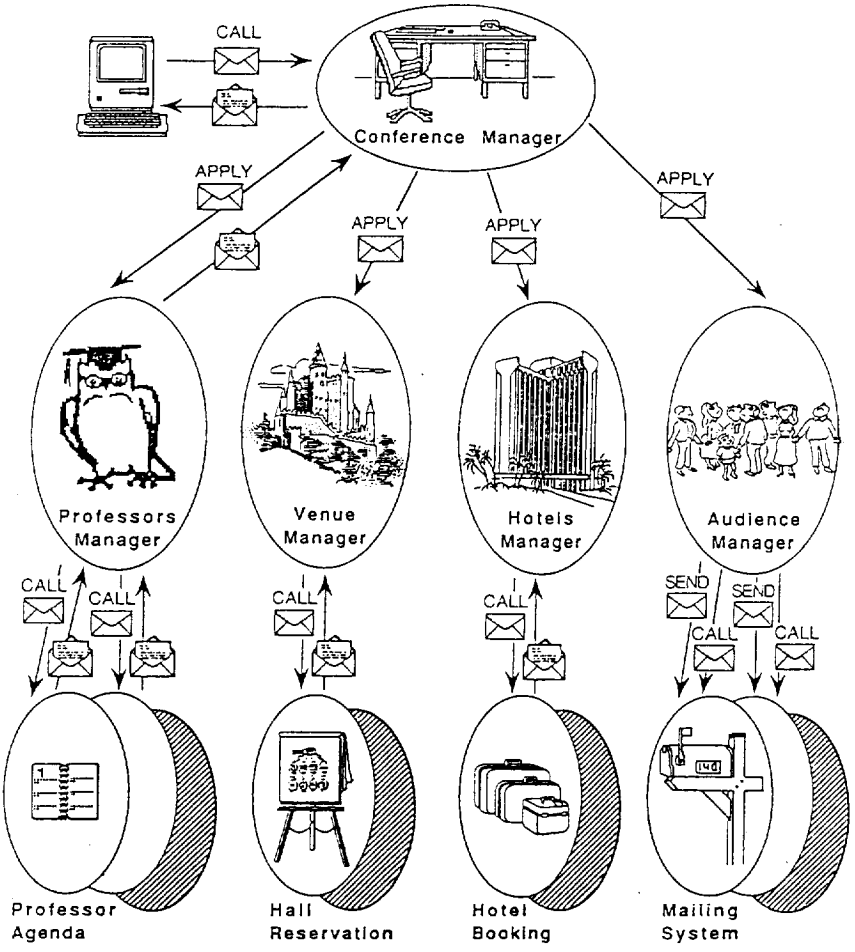


Fig. 6

At their turn, each of the "second level managers" creates other nested actions by the way of *Call* messages. These are *Call* messages because we suppose that if one of these subactions fails the client manager may choose to go on, e.g. trying to contact another professor or to book another hotel; the replies are used by each client manager for the same purpose (e.g. if the selected hall in the venue is already reserved the *vm* can send a message to another one). The messages to the invited guests Mailing Systems are invitations (sent through a *Call*) as well as other university-relevant information, e.g. activity reports (sent through a *Send* because they are independent from the conference organization task).

We point out that the effects of the *Call* and *Apply* messages are committed only if the top-level parent action terminates without failures, while the actions created by the *Send* messages are carried out independently.

Following are some significant parts of the methods involved in the conference organization example:

```
class ConferenceManager : public ActiveObject
{   ..............
public :
  method(organize_conference)
  {       ..................
  Int n_profs = Apply (pm, ProfessorsManager, contact_profs) <<
                                                    subject << date ;
  Apply (vm) <= Method (VenueManager, book_hall) << date ;
  Apply (hm, HotelsManager, book_hotel) << date << n_profs ;
  Apply (am, AudienceManager, invite_audience) << subject <<
                                                    venue << date ;
  ..................
  }
..................
};

class HotelManager : public ActiveObject
{   ..............
public :
  method(book_hotel)
  {       ..................
  Bool rep = Call (hb1, HotelBooking, book_rooms) << date <<
                                                    n_profs ;
  if ( (rep == 0) || (Failure (rep)) )
      rep = Call (hb2, HotelBooking, book_rooms) << date <<
                                                    n_profs ;
  ..................
  }
..................
};

class AudienceManager : public ActiveObject
{   ..............
public :
  method(invite_audience)
  {       .....................   /* guest_list is declared as an array
                                  of references to Mailing System objects */
  for ( i = 1 ; i < n_guests ; i++ )
    {
       Call (guest_list[i], MailingSystem, accept_invitation) <<
                                          subject << venue << date;
       Send (guest_list[i], MailingSystem, accept_information) <<
                                                activity_report ;
    }
  ..................
  }
..................
};
```

## 6 Concluding remarks

Global atomic actions relating objects executions are necessary in object-based distributed systems to protect applications consistency.
We believe that the best form of atomicity is the *implicit atomic service* semantics: a service is enclosed inside an atomic action and the logical nesting of services corresponds to the nesting of actions. This gives a powerful semantics to the remote invocation metaphor. As we have pointed out in Section 2, all the previous approaches providing *atomic service* semantics are based on RPC. Nevertheless, RPC is not always the best communication policy to take advantages of the available parallelism of distributed systems.

In this paper we presented an asynchronous object communication protocol, named ACS (Apply, Call, Send), that we designed and implemented within the KAROS concurrent object-oriented language.
The protocol is based on three different kind of *asynchronous message passing* providing *atomic service* semantics: each service requested by an object is either entirely executed or not executed at all, in which case its intermediate effects are not visible to concurrent activities. Furthermore, the effects of a top-level committed service are not undone by a node failure.
The *Apply* and *Call* messages provide two complementary ways of nesting actions, while the *Send* message allows to safely break the atomicity of an activity and thus to avoid unwieldy inefficiencies due to the failure atomicity and serialisability properties of an action.

We can not pretend to have solved the problem of achieving a clean integration between global action constructs and object-based distributed systems. In fact, mechanisms like two phase locking unnecessary restrict concurrency. Other algorithms, such as optimistic certifiers, provide more concurrency but increase the potential number of abortions and communications between objects. In fact, the best atomicity mechanism depends on the cost of communications and of actions aborts, as well as on the available concurrency in the system. An important issue that deserves further study is to design general atomicity mechanisms that can be adapted to specific situations along the line of histories [28].

## Acknowledgments

We sincerely acknowledge Jean Ferrié for reading and discussing earlier drafts of this paper.

## References

1. Bernstein P.A , Hadzilacos V and Goodman N - Concurrency Control and Recovery in Database Systems - Addison Wesley - 1987.
2. Birell A and Nelson B - Implementing Remote Procedure Call - ACM Transactions on Computer Systems - February 1984.
3. Capobianchi R, Guerraoui R, Lanusse A and Roux P - Active Objects on a Parallel Machine: A Case Study - Proc Conf on Technology of Object Oriented Languages and Systems - March/April 1992.
4. Caromel D - Concurrency : an Object-Oriented Approach - Proc Conf on Technology of Object-Oriented Languages and Systems - June 1990.

5. Corradi A and Leonardi L - The Specification of Concurrency : An Object-based Approach - Proc IEEE Conference on Computers and Communications - March 1988.
6. Cheriton D - The V Distributed System - Comm ACM - March 1988.
7. Dixon G.N, Parrington G.D, Shrivastava S.K and Wheater S.M - The Treatment of Persistent Objects in Arjuna - Proc European Conf on Object-Oriented Programming - July 1989.
8. Guerraoui R, Capobianchi R, Lanusse A and Roux P - "Une vue générale de KAROS : un langage à objets concurrents destiné à des applications distribuées" - Technical Report CEA, CE Saclay DEIN/SIR (in french) - January 1992.
9. Guerraoui R - "Customizing Concurrency Control in an Object-Oriented Framework" - Technical Report CEA, CE Saclay DEIN/SIR - March 1992.
10. Gray J.N - Notes on Database Operating Systems - Lecture Notes in Computer Science - Springer - 1978.
11. Kafura D - ACT++ : Buiding a Concurrent C++ With Actors - Journal of Object-Oriented Programming - May/June 1990.
12. Kaiser G.E, Popovitch S.S, Hseush W and Wu S.F - MELDing Multiple Granularities of Parallelism - Proc European Conf on Object-Oriented Programming - July 1989.
13. R.H Halstead - Multilisp : A Language for Concurrent Symbolic Computation - Transactions on Programming Languages and Systems - October 1985.
14. Ishikawa Y and Tokoro M - Orient84/K : An Object-Oriented Concurrent Programming Language for Knowledge Representation - Object-Oriented Concurrent Programming - ed A Yonezawa and M Tokoro, The MIT Press - MA, 1987.
15. Leblanc R.J and Wilkes C.T - Systems Programming with Objects and Actions - Proc IEEE Conf on Distributed Computing Systems - 1985.
16. Liskov B and Sheifler R - Guardians and Actions: Linguistic Support for Robust, Distributed Programs - ACM TOPLAS - July 1983.
17. Meyer B - Object-Oriented Software Construction : ed Prentice-Hall - 1988.
18. Moss J.E.B - Nested Actions : an Approach to Reliable Distributed Computing - Ph.D thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA, 1981.
19. Moss J.E.B - Nested Transactions And Reliable Distributed Computing - Proc Symposium on Reliability in Distributed Software and Database Systems - July 1982.
20. Nierstrasz O.M - Active Objects in Hybrid - Proc Conf on Object-Oriented Programming, Systems, Languages and Applications - Orlando, October 1987.
21. Calton P and Jerre D.N - Design and Implementation of Nested Transactions in Eden - Proc Symp on Reliability in Distributed Software and Database Systems - March 1987.
22. Schantz R, Thomas R and Bono G - The Architecture of the CRONUS Distributed Operating System - Proc IEEE Conf on Distributed Computing Systems - May 1986.
23. A.H Skarra, S Zdonik - Concurrency Control and Object-Oriented Databases - in Object-Oriented Concepts, Databases, and Applications - ed W Kim and F.H Lochovsky, Addison Wesley, 1989.
24. Stroustrup B - The C++ Programming Language - Addison Wesley - 1986.
25. Tony P.Ng - Using Histories to Implement Atomic Objects - Transactions on Programming Languages and Systems - November 1989.

26. Walker E, Floyd R and Neves P - Asynchronous Remote Operation Execution in Distributed Systems - Proc IEEE Conf on Distributed Computing Systems - May 1990.
27. Weihl W.E - Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types - Transactions on Programming Languages and Systems - April 1989.
28. Yokote Y and Tokoro M - Concurrent Programming in Concurrent Smalltalk - Object-Oriented Concurrent Programming - ed A Yonezawa and M Tokoro, The MIT Press - MA, 1987.
29. Yonezawa A, Shibayama E, Takada T and Honda Y - Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1 - Object-Oriented Concurrent Programming - ed A Yonezawa and M Tokoro, The MIT Press - MA, 1987.