`Multithreaded Rendezvous`: A Design Pattern for Distributed Rendezvous*

Ricardo Jiménez-Peris
Marta Patiño-Martínez
Sergio Arévalo

Universidad Politécnica de Madrid
Facultad de Informática
28660 Boadilla del Monte (Madrid), Spain
{rjimenez, mpatino, sarevalo}@fi.upm.es

Keywords: Design Patterns, Rendezvous, Multithreaded servers, Distributed Systems, Object-Oriented Technologies.

ABSTRACT

In this paper we describe a design pattern for distributed rendezvous. We propose a variant of rendezvous that supports multiple server threads, each one devoted to a different client. On the server side a `ForwarderObject` is in charge of forwarding calls to the corresponding servers threads. This design pattern encapsulates both the forwarding algorithm and the server interface, so both can be changed independently. Guidelines are given on how to implement the design pattern in Ada 95, taking advantage of language specific features such as streams. The Multithreaded Rendezvous pattern has been successfully applied in the implementation of Transactional Drago, an Ada extension to program fault-tolerant distributed applications.

1 INTRODUCTION

Rendezvous has been widely used to synchronize concurrent programs. In a concurrent program the server offers a set of services that clients call. In order to provide those services the client and server must synchronize, that is, the client should ask for a service and the server must be ready to execute it. Although rendezvous is provided as a basic mechanism in some languages, like Ada, distributed rendezvous is not. Using it in a distributed environment will be natural, especially for those that are used to work with it to build concurrent programs, but also to transform concurrent applications based on rendezvous into distributed ones.

In applications based on rendezvous, when a client requests a service to a server, it issues a call to an entry point of the server with some parameters and then blocks till the server sends a reply with the service results. On the other hand, the server has its own flow of execution and at some points it accepts requests. When the server wants to accept a request for a particular service, it will block till a request for that service is received from a client. After that, the server executes the service, sends the results back to the client and then continues execution. Thus, rendezvous allows imposing a protocol of calls to clients. Distributed rendezvous implies that requests and replies may cross the network from the client node to the serve node.

The implementation of distributed rendezvous (fig. 1) is very similar to the implementation of the well-known RPC model [3]. In this implementation, when a server is designed, its services (or entry points) are specified by their name, and information about their parameters (types and modes) is also provided. From that specification, client and server stubs are automatically generated. The server stub is linked with the server and the client stub is linked with any client willing to use that server.

The client stub must offer exactly the same interface the server offers. The entry point implementation in the client stub must be able to locate the server, send it a message with the parameters of the call (flattened). Then, it waits for the server reply. When it is received it unflattens the results and returns them to the client. Thus, the client makes calls to the remote server as it they were local.

The server stub will have its own flow of control that receives messages from clients and then finds out which entry point is the message aimed to. It creates a task that knows how to unflatten the message and how to call the entry point. This kind of tasks represents the client on the server side. When the server finishes the service, it returns the result to that task and the task sends a message back to the client stub with the flattened results.

When clients and servers interact by means of rendezvous, servers accept calls from different clients in their code. However, there are situations where it is not interesting that the same server accepts interleaving calls from different clients. For instance, in a transac-

*This work has been partially funded by the Spanish Research Council(*CICYT*), contract number *TIC98-1032-C03-01* and the Madrid Regional Council (*CAM*), contract number *CAM-07T/0012/1998*.

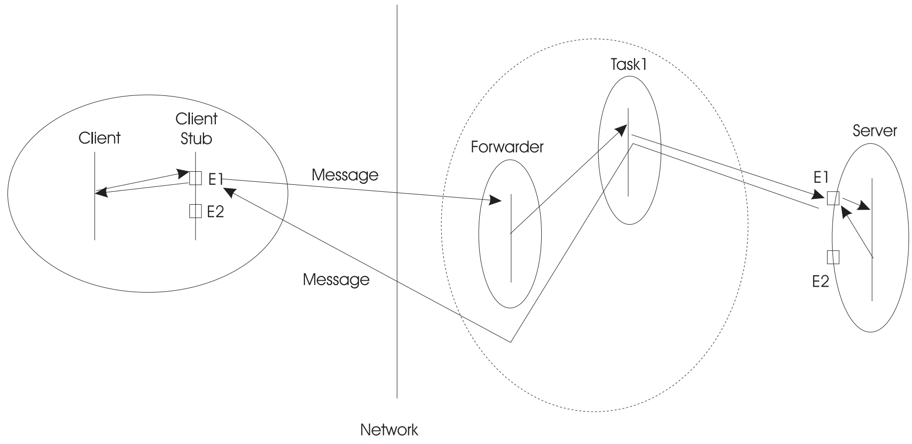

Figure 1: `Multithreaded Rendezvous` Pattern

tional setting, services are called from different transactions. In order to prevent transactions from seeing uncommitted results from other transactions, calls must be processed by different threads. Thus, sometimes it is adequate to extend the semantics of rendezvous as follows: every time a new client calls a server, a new thread with the code of the server is created. In this way, the server code will deal with a single client, reducing the complexity of the server. This approach is taken in [9].

In this paper it is presented `Multithreaded Rendezvous`, a design pattern for distributed rendezvous, and guidelines to implement it in Ada 95. This design pattern has been used in the implementation of the language *Transactional Drago* [8], an Ada extension to build fault-tolerant distributed applications that offers rendezvous as synchronization mechanism for distributed transactions [9]. Next section presents the `Multithreaded Rendezvous` design pattern and a suggested implementation in Ada 95. Section 3 presents its application in *Transactional Drago*. Related design patterns are presented in section 4 and finally, and some conclusions in section 5. In the rest of the paper it is assumed basic knowledge of Ada 95 and design patterns.

2 THE `Multithreaded Rendezvous` DESIGN PATTERN

In traditional rendezvous, servers are single-threaded, that is, a single flow of control processes the requests from all the clients. In our approach the server is multithreaded, that is, there is a thread per client, so it is needed an object to manage the different server threads. This object will accept messages sent to the server and will forward them to the corresponding server thread. It will also be in charge of creating server threads as needed. We will call it forwarder object, as it forwards messages to the corresponding server thread. As all the calls go through the forwarder, it can apply forwarding policies. For instance, it can forward calls as soon as they arrive or forward them sequentially (i.e. until a call is not processed, the next call is not forwarded). The proposed design pattern will deal with the following elements: calls, flattening (marshalling) and unflattening (unmarshalling) of parameters, communication layer, forwarders and server threads.

Additionally, the `Multithreaded Rendezvous` design pattern aims the following goals:

- To encapsulate the forwarding policy. Thus, new forwarding policies can be introduced without changing existing servers.
- To encapsulate server interfaces and types. At the same time, the forwarder must be able to create server tasks of the appropriate type and store references to them.
- The forwarder must be able to identify clients from calls. This requirement and the previous one will allow adding new kinds of servers reusing existing forwarders.
- To encapsulate the communication layer, so it can be changed without disturbing clients and servers.
- To ensure type checking of server calls when the client is compiled.
- To minimize the effort of writing new servers, reducing the extra code to be written. In this way, server code can be written without any auxiliary tools, such as stub generators, etc.
- The server code should be the same as if it were not distributed, so it can use linguistic mechanisms to receive the calls (e.g. select statement, count attribute, etc.).

In the next subsections, we will motivate and describe the different classes participating in the design pattern.

2.1 The `StubObject`

When a client calls a service, the call and the parameters must be flattened to travel across the network. They must contain information about which server entry is called as well as `in` and `in out` parameters. When calls are processed they must return the `out` and `in out` parameters. We make use of Ada 95 streams [4, 1] to flatten the parameters as the implementation language, Ada 95, will help us with this task, providing default `Input` and `Output` routines for every declared type, for flattening and unflattening data.

As the activity of flattening the call, send it across the network, wait for the reply and unflatten the reply must be done for every call, we will encapsulate it in a `StubObject`. The `StubObject` will need two interfaces one for the client side and another one for the server side. It could be split into two different classes but this approach does not offer new advantages, whilst it will increase the complexity of the system. The following interface will suffice:

```
type StubObject is abstract tagged private;
type StubObjectAccess is
   access all StubObject'Class;

type KindOfStubType is (local, remote);

-- Initializes a Stub as local or remote.
procedure Init(
   stub : out StubObject;
   kind : in  KindOfStubType:= local );

-- Flattens the call and sends it to the
-- remote site. It waits until the reply
-- is received, and then unflattens the
-- call and returns the results. This
-- method is used only on the client side.
procedure Call(
   stub  : in StubObject;
   aCall : in out CallObjectAccess );

-- Receives a call from a client and returns
-- the call unflattened.
procedure Accept(
   stub  : in StubObject;
   aCall : in out CallObjectAccess );

-- Flattens the result of the call and sends
-- back the reply to the client.
procedure Reply(
   stub  : in StubObject;
   aCall : in CallObjectAccess );
```

In order to ensure type checking on the client side, concrete `StubObjects` are introduced for each server. That is, a concrete server stub will only receive calls expected by that server (see `StubX` and `StubY` in fig. 4). It must be noticed that the behavior of concrete classes methods will be the same than in the abstract class, for this reason they just propagate the call to the inherited method of `StubObject`, but their interface is stricter (i.e. `StubX` methods will not accept parameters that do not derive from `ServerXCall`).

2.2 The `Comm` Class

One of our goals is to encapsulate the communication layer so it can be changed without disturbing clients and servers. We will implement it as a stream, thus it integrates smoothly with the flattening and unflattening process (by means of the `Input` and `Output` operations). This is achieved with the following class:

```
type Comm is abstract new Root_Stream_Type
   with private;
type Stream_Access is access 'Class;

-- Creates a Comm object associated to a
-- network address.
procedure Init(
   aComm   : out Comm;
   address : in String );

-- Send a request to the server (client side
-- method). A call must be flattened (with
-- Output) before calling this method.
procedure SendRequest(
   aComm   : in out Comm );

-- Receive a request from a client (server
-- side method). Then the call can be
-- unflattened with Input.
procedure ReceiveRequest(
   aComm   : in out Comm );

-- Send a reply to a client (server side
-- method). A reply call must be flattened
-- (with Output) before calling this method.
procedure SendReply(
   aComm   : in out Comm );

-- Receive a reply from the server (client
-- side method). Then the reply call can be
-- unflattened with Input.
procedure ReceiveReply(
   aComm   : in out Comm );
```

2.3 The `CallObject`, `ForwarderObject`, `CourierTask` and `ServerTaskObject`

Clients and the `ForwarderObject` will interact by means of `StubObjects`. However, there is a problem to be solved. The forwarder receives calls from clients, but as it will deal with different servers, it must not know the server interface neither its type, thus servers can be added without modifying the forwarder. Then, how can it forward a call to a server? On the other hand, the client knows the server interface, and knows which entry it wants to call, so we need to encapsulate this knowledge in some class and let this class to make the call to the server thread. With this aim in mind we have created an abstract class `CallObject` from which concrete calls will inherit. However, this is not a trivial task and there are still some problems to solve:

1. The concrete `CallObject` will know the server interface, which entry to call and with which parameters, but it will not know which concrete task to call, as this is only known by the forwarder.

2. On the other hand, the forwarder does not know the server interface nor which call to make.

3. The forwarder must be able to create server tasks but without knowing their type. And what it is more, it must store references to them to be able

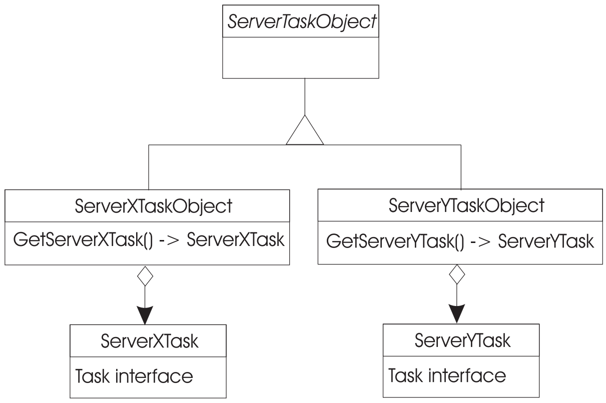

Figure 2: `ServerTaskObject` Hierarchy

to forward them future calls and apply forwarding policies.

4. The forwarder must be able to identify clients from calls in order to be able to associate them to the right server thread.

The first problem is solved providing `CallObject` with a method to set the destination task. The forwarder will call this method to inform the `CallObject` which server task to call.

The second one is solved by letting the concrete `CallObject` to make the call. It must provide a method to call the destination task previously set. After the forwarder has set the destination task, it will invoke the `MakeCall` method, that in turn it will call the entry of the server task.

The third problem is solved converting the `CallObject` into an `Abstract Factory` [5] of server tasks. The `Abstract Factory` design pattern allows a client to be independent of how concrete products are created and represented. We achieve this by providing `CallObject` with a method to create a server task, `StartServerTask`. It must be noted that we are using the concrete `CallObject` tag (that it is known, due to Ada streams are typed) to create a `TaskServerObject` of the appropriate type, that is, to redispatch to the right `StartServerTask`.

The fourth problem is solved by providing the `CallObject` with two methods one to set the client identifier (`SetID`) and another consult it (`GetID`). Then, the forwarder can ask for the client identifier corresponding to a call (as the concrete `CallObject` is the only that has such knowledge), in order to decide to which server thread associate the call.

Another detail to be solved is related to task typing in Ada. Ada tasks are not tagged types, but just limited types. Thus, it is not possible to derive task interfaces from a root type. We have overcome the problem by creating an abstract root null class, `ServerTaskObject` from which concrete classes inherit (fig. 2) and extend its state with a particular server task. As this abstract class is known by the forwarder, it can store references to `ServerTaskObjects`. Concrete classes extending `ServerTaskObject` will provide a method to obtain a pointer to the contained task, in order to be called by some object that knows its interface. In our pattern, the object that knows the server task interface is the concrete `CallObject` corresponding to such a server.

Taking into account all the problems and the proposed solutions we have devised the following interface for the `CallObject` class:

```
type CallObject is abstract tagged
   limited private;
type CallObjectAccess is access all
   CallObject'Class;

-- Creates a ServerTaskObject.
function StartServerTask(
   aCall : CallObject )
      return ServerTaskObjectAccess is abstract;

-- Sets the server task to be called.
procedure SetDestinationTask(
   call   : in out CallObject;
   server : in ServerTaskObjectAccess );

-- Gets the client identifier.
procedure GetID(
   call : in CallObject;
   id   : out ClientIdentifier );

-- Sets the client identifier.
procedure SetID(
   call : in out CallObject;
   id   : in ClientIdentifier );

-- Call the service requested by the client in
-- the previously set server task.
procedure MakeCall(
   call : in out CallObject ) is abstract;

-- Flattens the call. The object knows in which
-- side is (client or server side). It will
-- flatten in parameters on the client side and
-- out parameters on the server side.
procedure Output(
   stream : access

      Ada.Streams.Root_Stream_Type'Class;
```

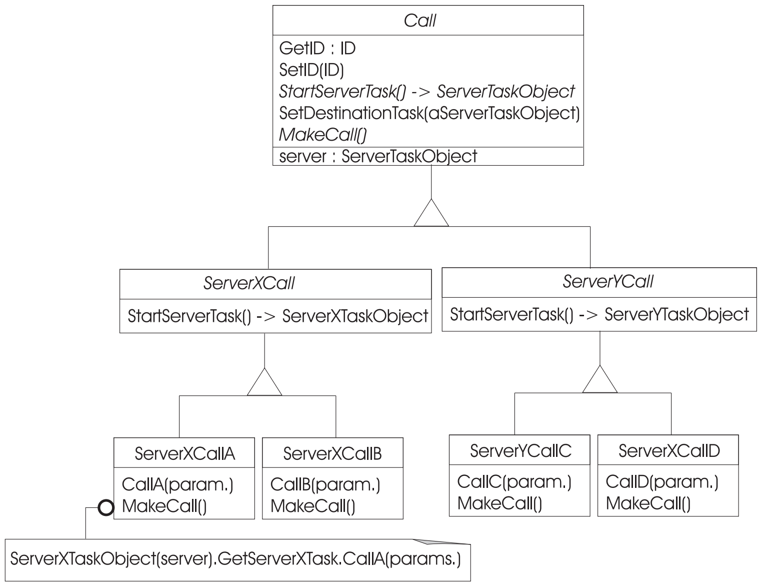

Figure 3: `Call` Hierarchy

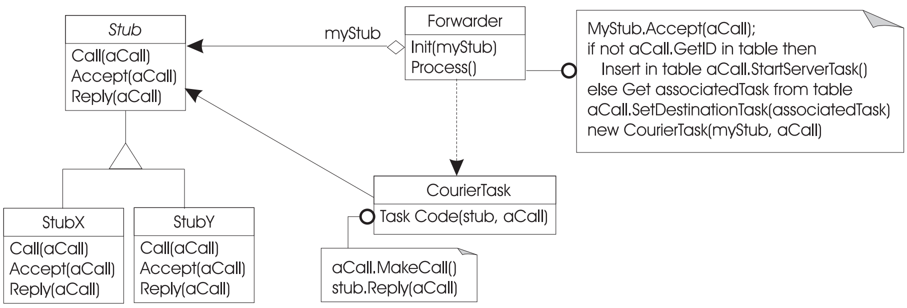

Figure 4: Forwarder and Stub

```
   obj    : in CallObject );

-- Unflattens the call. The object knows in which
-- side is (client or server side). It will
-- unflatten in parameters on the server side and
-- out parameters on the client side.
procedure Input(
   stream : access
      Ada.Streams.Root_Stream_Type'Class;
   obj    : out CallObject );
```

When instantiating `CallObject` for a concrete server it is convenient to do it in two steps (see `Call` hierarchy in figure 3). The reason is that the `StartServerTask` method is the same for all the `CallObjects` of a concrete server. First, it can be overridden in a class that is still abstract, `ServerXCall`; second, from this class the concrete `CallObjects` can be derived overriding the `MakeCall`, `Input` and `Output` methods.

The `ForwarderObject` has a very simple interface. Forwarders are initialized with a concrete `StubObject`. The `Process` method processes a message, that is, accepts the message from the stub and forwards it to the appropriate server task if it exists. If the server task does not exists, it creates a new one, storing a reference to it to forward future calls to it. In order to prevent the blocking of the `ForwarderObject` during the execution of calls, the `ForwarderObject` creates an auxiliary task to actually make the call, a `CourierTask`. This task has two parameters, the `CallObject` (whose `MakeCall` method must call) and the `StubObject` to reply to the client. This is the Ada 95 interface of the `ForwarderObject`:

```
type ForwarderObject is abstract tagged private;
type ForwarderObjectAccess is access
   ForwarderObject;

-- Creates a forwarder, associating it
-- a server stub.
procedure Init(
   forwarder : out ForwarderObject;
```

```
   my_stub : in StubObjectAccess );

-- Accepts a message and forwards it to the
-- appropriate server task. If it does not
-- exist, then it is created.
procedure Process(
   forwarder : in ForwarderObject );

task type CourierTaskType(
   stub : StubObjectAccess;
   aCall : CallObjectAccess) is
end CourierTaskType;

type CourierTaskAccessType is access
   CourierTaskType;
```

2.4 Putting all Together

Every time a server X is added, the server programmer must program just a `ServerXCall` hierarchy, as well as the corresponding `ServerXTaskObject`. The programmer will also derive a concrete `StubObject` whose method parameters will be `ServerXCalls`. Their code will just call their corresponding inherited methods of the root abstract class `StubObject`. The server main program will just instance a concrete `ForwarderObject` and associate a `StubObject` for server X to it. However, the programmer can be freed from the last to tasks (writing a concrete stub object and main program) by creating two generic packages.

Adding a new `ForwarderObject` is even simpler. It suffices to write a new concrete forwarder and instantiate it in the server main program where we want to use it. A client program will just import the appropriate package where the `ServerXCall` hierarchy and the `StubXObject` are defined. A call to a service of server X will be like:

```
Call(myStubX, CallA(params));
```

Now, it is possible to compose all the pieces of the system. Figures 2, 3 and 4 show the class hierarchies and their relationships. To recapitulate, in figure 6 it is shown the complete interaction among object instances when a call is made and the server task for the client is already created.

Notice that the concrete `CallObject` (`aServerXCallA`) represented in the interaction diagram (fig. 6) represents two different instances distinguished by the brackets, one on the client side (the outer one) and other on the server side (the inner one). The instance on the server side has a copy of the state (including the tag) of the one on the client side. In the figure, it has been represented as a single object because logically it is a single object that travels from the client side to the server side and when the call completes returns to the client side. The interaction would be slightly different when a client calls for the first time to the server, as the forwarder will first create a server task by calling the `StartServerTask` method of the call object.

2.5 The `Call` Path

To summarize we describe the whole path of a call (fig. 5):

1 The client makes the request to the client stub.

2 The client stub flattens the call and submits it using the comm object.

3 The flattened call crosses the network.

4 The server stub gets the call from the comm object and unflattens it.

5 The forwarder takes the call from the local stub. If a server thread does not exist for the client, it creates a new one.

6 The forwarder creates a courier task to make the call asynchronously.

7 The courier task makes the call and the server thread accepts it (rendezvous).

Once the call is processed the result returns through this path:

a The rendezvous between the courier task and the server thread finishes.

b The courier task returns the results of the call to the server stub.

c The server stub flattens the call and sends it back by means of comm.

d The returned call with the results cross the network.

e The client stub unflattens the call.

f The client unblocks and gets the call results.

3 APPLYING THE `Multithreaded Rendezvous` DESIGN PATTERN TO *TRANSACTIONAL DRAGO*

Transactional Drago [8] is an Ada and *Drago* [7] extension to program fault-tolerant distributed applications. It implements the group transaction model, an integration of the group and nested transaction paradigms. Groups are made up of agents, the unit of distribution in *Transactional Drago*. There are two kinds of groups: cooperative and replicated. Cooperative groups permit programmers to express parallelism and so increase throughput. Replicated groups allow for the programming of fault-tolerant applications according to the active replication model. Agents are similar to Ada tasks, they offer a set of remote entries and that are accepted (by means of an `accept` statement) in the agent code. Agents of a group are usually located at different nodes of the system to speedup the services of that group and at the same time provide fault tolerance. Agents communicate using distributed rendezvous. Group calls are multicasted to all the agents of the server group; the agents will eventually accept the call, execute the service and return the result and control to the caller. When several client transactions call the same group, the calls are not accepted by the same thread. A new thread in each agent will be created to accept all the calls from each client transaction in order to prevent transactions from seeing uncommitted results of other transactions.

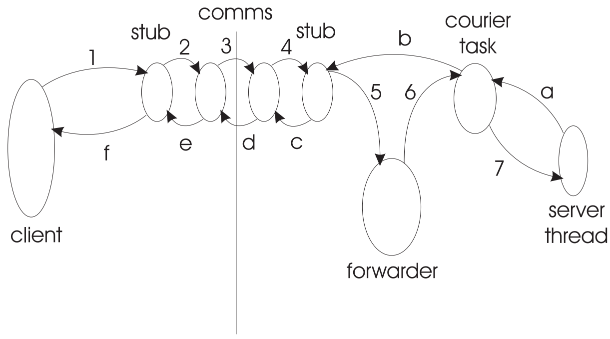

Figure 5: `Call` Path

Transactional Drago is preprocessed to Ada 95 and service calls from two libraries: *GroupIO* [6] and *TransLib*. The former provides support for group communication and the latter for transaction processing. To use the `Multithreaded Rendezvous` design pattern it is necessary to provide a concrete `Comm` class, concrete `CallObject` classes for the different concrete `ServerObject` classes, concrete `ServerTaskObjects` and concrete `ForwarderObjects`.

Let's start with the `Comm` class. A server in *Transactional Drago* is a group of agents. Clients call a server by multicasting the call to all the agents of the group. The `Comm` class is a stream, that encapsulates the multicast communication layer provided by *GroupIO*. *GroupIO* only deals with flattened objects, thus the `Multithreaded Rendezvous` design pattern simplifies the marshalling and unmarshalling of the parameters.

In *Transactional Drago* clients are identified by the transaction identifier (tid) of the transaction enclosing the call to the server. When a concrete `CallObject` is created it can know the tid of the client transaction without forcing the client to set it (calling to `SetID`). We have used the task attributes facility provided by the programming systems annex that allows to associate state to tasks and to consult it. So, tasks in *Transactional Drago* have their state extended with the tids of the transactions they are currently executing. In this way, the call object can find out and record which transaction is issuing the call, so on the remote side the forwarder will be able to know to which server thread forward the call.

Two different kinds of forwarders (i.e. forwarding policies) are used in *Transactional Drago*. One for cooperative groups and other for replicated groups. The forwarder for cooperative groups forwards messages as they arrive in order to achieve the maximum concurrency. However, the forwarder for replicated groups must guarantee replica determinism. For this reason, it forwards messages one by one, that is, until a message has not been processed it does not forward the next one. If the forwarder for cooperative groups were used, and the replicas executed a `select` statement different replicas could accept any of the forwarded messages, thus breaking the determinism.

4 RELATED WORK

There are some design patterns described in [5] related to our design pattern. The `CallObject` class together with the method `MakeCall` can be considered an application of the `Command` design pattern. The intent of the `Command` design pattern is to encapsulate requests as objects, so they can be used by other objects that do not know anything about the operation requested or the receiver of the request. Our `CallObject` could be assimilated to `Command`, `MakeCall` to `Execute`, `Forwarder` to `Invoker`, and `ServerTask` to `Receiver`. However, we have had to solve additional problems that do not appear in the `Command` design pattern. In particular, the `Forwarder` has to know something about the receiver of the operation, as it has to forward the call to a particular server thread, while in `Command` the `Invoker` does not know anything about the `Receiver`. Another important difference is that `Invoker` is called by the `Client` in `Command`, while in `Multithreaded Rendezvous` the Client and `Invoker` are both active at the same time (the client application and the server process).

The `Forwarder` and its `Process` method can be seen as an instance of the `Strategy` pattern (assimilating the `Forwarder` to `Strategy` and `Process` to `AlgorithmInterface`), where the `Strategy` encapsulated is the `Forwarding Policy`.

The `CallObject` class and its method `StartServerTask` can be seen as an instance of the `Abstract Factory` pattern as they allow the `Forwarder` to rely on a concrete `CallObject` (a concrete factory) to create a concrete product of the `ServerTaskType`.

In [2] it is proposed a programming paradigm for synchronizing multiple clients and servers, showing how to forward on entry calls combining class-wide types and protected objects.

5 DISCUSSION AND CONCLUSIONS

The `Multithreaded Rendezvous` design pattern can be applied to very different situations where there is a frontier to cross like a network, different address spaces in the same site (e.g. the operating system kernel and the user application spaces, etc.).

The proposed design pattern can also be used to implement the coordinator/workers model.

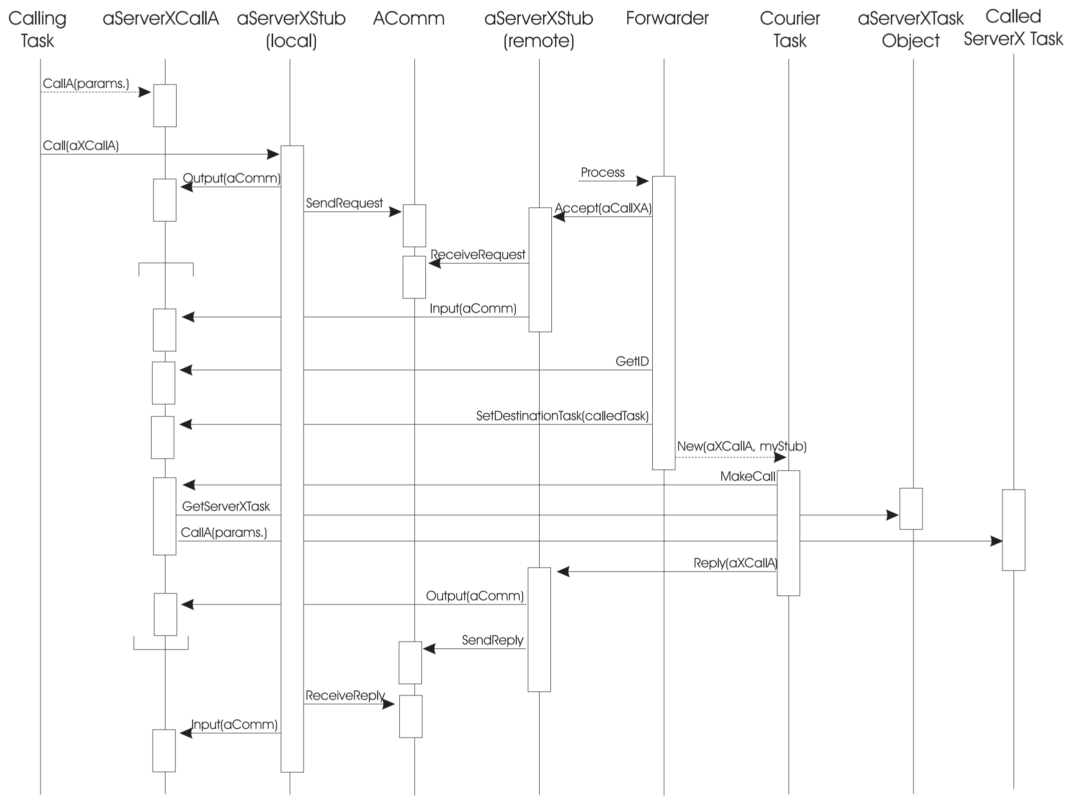

Figure 6: `Process` Call Interaction Diagram

The coordinator role will be represented by the `ForwarderObject`. It will distribute work among their workers (`ServerTaskObjects`) and it will also create workers as needed. The `GetID` method abstracts the information needed from the call to distribute the work. In addition, the design pattern slightly modified can be used in scenarios where a single `ForwarderObject` distributes calls for different servers. The key difficulty for this application is dispatching calls to the right server task (creating them as needed), and it is already solved in the proposed design pattern.

In this paper the `Multithreaded Rendezvous` design pattern and some implementation guidelines for Ada 95 have been presented. The `Multithreaded Rendezvous` design pattern is useful to build distributed client-server applications whose communication model is rendezvous. Where different server processes are created for each client and each server executes a set of services for a particular client. The `Multithreaded Rendezvous` design pattern encapsulates both the forwarding policy and the server task interface, so both can be modified independently, thus new servers can be added reusing existing forwarders. The communication layer is encapsulated to allow any communication protocol without disturbing existing clients and servers. Server calls in client code are type safe as the client is compiled against server code. The task of adding servers is light enough to be done at hand without stub generators. Server tasks are programmed as if they were to be called locally. It has also been shown an application of the `Multithreaded Rendezvous` design pattern in the implementation of *Transactional Drago*.

References

[1] M. Ben-Ari. *Ada for Software Engineers*. John Wiley, 1998.

[2] M. Ben-Ari. Synchronizing Multiple Clients and Servers. In L. Asplund, editor, *Proc. of Conference on Reliable Systems. Ada-Europe'98*, volume LNCS 1411, pages 41–52. Springer, June 1998.

[3] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[4] N. H. Cohen. *Ada as a Second Language*. McGraw-Hill, 1996.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[6] F. Guerra, J. Miranda, Á. Álvarez, and S. Arévalo. An Ada Library to Program Fault-Tolerant Distributed Applications. In K. Hardy and J. Briggs, editors, *Proc. of Reliable Software Technologies, Ada-Europe'97*, volume LNCS 1251, pages 230–243. Springer, 1997.

[7] J. Miranda, Á. Álvarez, S. Arévalo, and F. Guerra. Drago: An Ada Extension to Program Fault-tolerant Distributed Applications. In A. Strohmeier, editor, *Proc. of Reliable Software Technologies, Ada-Europe'96*, volume LNCS 1088, pages 235–246. Springer, 1996.

[8] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In L. Asplund, editor, *Proc. of Int. Conf. on Reliable Software Technologies, Ada-Europe'98*, volume LNCS 1411, pages 78–89. Springer, June 1998.

[9] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Synchronizing Group Transactions with Rendezvous in a Distributed Ada Environment. In *Proc. of ACM Symposium on Applied Computing*, pages 2–9. ACM Press, Feb. 1998.

Ricardo Jiménez-Peris holds a MS degree in computer science from Universidad Politécnica de Madrid (1990). He is currently working on *TransLib*, an object oriented framework to program fault-tolerant programming systems. He has been assistant professor in computer science at Universidad Politécnica de Madrid since 1990. He is member of the Association for Computing Machinery.

Marta Patiño-Martínez received a MS degree in computer science in 1990 from Universidad Politécnica de Valencia. She is currently working on *group transactions*, an integration of the transaction and group communication paradigms. She has been assistant professor in computer science at Universidad Politécnica de Madrid since 1990. She is member of the Association for Computing Machinery and the IEEE Computer Society.

Sergio Arévalo is an associate professor of computer science at Universidad Politécnica de Madrid. His research interests include distributed systems, fault-tolerance, operating systems and real-time systems. He has been visiting researcher at ATT Bell Laboratories and research fellow in the European Space Agency. He received a PhD in Computer Science from Universidad Politécnica de Madrid. He is a member of ACM since 1983.