Implementing Replicated Services in Open Systems Using a Reflective Approach

Joni Fraga, Carlos Maziero, Lau C. Lung, Orlando G. Loques Filho[*]

Laboratório de Controle e Microinformática
Departamento de Engenharia Elétrica - UFSC
88.049-900 Florianópolis SC - BRAZIL
e-mail: {*fraga,maziero,lau*}*@lcmi.ufsc.br*

[*]Pós-Graduação em Computação Aplicada e Automação - UFF
24210-240 Niterói RJ - BRAZIL
e-mail: *loques@caa.uff.br*

Abstract

In this paper we evaluate the use of an object-oriented open platform based on the CORBA standard [15] for the implementation of replicated services. To improve the flexibility of the implementation, we use a reflective approach [13], which allows for separation of aspects related to the replication model from those related exclusively to the service being replicated. This separation makes it possible to modify the replication protocol according to the fault tolerance level desired, without any implications for the application code.

Keywords: fault tolerance, object groups, CORBA, computational reflection.

1. Introduction

Distributed systems have been recently characterized by their increase in dimensions and their heterogeneity. These systems have adopted the idea of open architecture, obtaining the interoperability of their components by the homogeneity of their corresponding interfaces. An effort made in terms of open programming is the CORBA standard (Common Object Request Broker Architecture), the result of the work of various companies which are part of the Object Management Group [15], whose aim is the integration of different programming systems based on objects. The use of CORBA standards, therefore, permits the interaction of objects distributed in the system, regardless of their coding languages, machine architecture or operational systems.

The concept of group processing has been introduced in distributed programming models with the aim of providing support for cooperative work (groupware), making possible an increasing availability of shared resources, or in replicated processing, due to fault tolerance. The use of CORBA standards has evolved in recent years in terms of incorporating group processing services. The Group Server abstractions are object of specification for inclusion in the OMA architecture [2]. The purpose of Group Server is similar to the approach used in ANSA (Advanced Networks Systems Architecture) [1], presenting a concentrator element in group communications, which is a handicap in the performance and reliability of a system.

Furthermore, various prototypes and even products of CORBA platforms have been developed, offering support to group processing. Specifically, we may mention the ORBs (Object Request Brokers) RDO/C++ [11], Orbix+Isis [10] and Electra [14]. These platforms make use of tools such as Isis [5] and Horus [18] that provide group communication based on the reliable broadcast concept. The tools cited above offer more reliable bases than the solutions sought in the specifications of Group Server in OMG.

In this article, we have set out to present our work on the integration of replication techniques into an open system, according to the patterns of the CORBA proposal, in order to make available mechanisms of fault tolerance to the applications distributed on that platform. The implementation of replication techniques is backed by the use of ORBs presenting a support for group processing.

With the aim of minimizing the replication reflexes on the programming of applications, a programming model was adopted, based on the *computational reflection* [13]. This paradigm permits the complete separation of the coordination mechanisms among the replicas from the application in itself. This separation, besides simplifying the programming of the replicated application, introduces a great flexibility into the system by allowing the alteration of the replication protocols, without interfering with the application functionality, or even involving changes on the level of execution support, which would be difficult, considering the nature of open systems.

The programming model presented was used successfully in the integration of different replication techniques. As an implementation support, use was made of Electra, an ORB with support to process groups. In the present article, we will present only the active competitive replication technique described in [17], to illustrate the advantages offered by this model in the environment under consideration.

The article is structured as follows: in section 2 we present the active competitive replication model; in section 3 we introduce the concepts of computational reflection and set out to structure the model according to this approach; in section 4 we describe the CORBA standard and the ORB Electra with its extensions for group support; finally, in section 5, we present in detail the integration of the reflective model proposed with the CORBA platform utilized and the results obtained in its implementation.

2. Software component replicated models

Replication techniques are an alternative that enables services to continue in distributed systems, even when failed nodes are present. The unit of replication is a software component (objects, processes, etc.), encapsulating data identified as *replica state*. The replicas are distributed among different sites in the network. The coordination of the replication defines the way the different replicas must interfere in the processing, in terms of maintaining the consistency and transparency of the whole.

The techniques vary according to the degree of synchronism and the types of replicas involved. In the literature, passive, active and semi-active replication models are identified [17]. In the passive replications, a privileged replica executes the processing referring to the input data, while the others have their states updated by the privileged one, using *checkpointing* (state transfer mechanisms). The *coordinator-cohort* model, presented in [4] is an example of this type of replication.

In active replication models, all the components receive the input data, process them simultaneously and produce the same outputs. In these models, identified as *State Machine*, the consistency of the replica state necessarily implies *determinism of replica*, which can be obtained by consensus about the input data and its order [19]. Some authors identify semi-active replications, in which, although all the replicas work in competition, only one produces the output. The order of the inputs is imposed by a privileged replica. The *leader-followers* technique described in [17] is an example of semi-active replication.

In [12] exhaustive studies are carried out on replication techniques and their implementation aspects. In this text we limit ourselves to the active competitive replication model described in [17].

2.1 Active competitive replication

In the competitive replication model all the replicas are active but only one responds to a given input data request. The main characteristic of this model is the competition among the replicas: only the fastest replies to the request. The coordination of the technique is distributed: each replica has an associated controller, responsible for receiving, broadcasting and comparing messages, with the corresponding replica dedicated to request processings. To guarantee replica consistency, all the messages among them are transmitted by means of atomic broadcasts. The competitive replication model can be devised so as to tolerate two sets of faults [17]: *timing faults*, involving semantics of *crash*, omission and timing errors; *arbitrary faults*, that take in the whole spectrum of failure semantics. For clarity and economy of space, we shall limit ourselves, in this text, to the first set of faults.

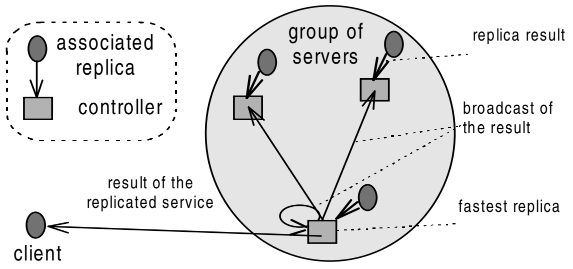

figure 1: Competitive replication model for temporization faults.

Figure 1 illustrates competitive replication, in a simplified manner, under the assumption of timing faults. In this case, considering the client/server model using a replicated server, a client request broadcast in the server group is received by the controllers, which send it to the corresponding replicas. On receiving the result of a processing of its associated replica**,** each controller verifies whether it has already received the message with the result of the same processing from another controller of the group. In the absence of these messages, the controller concatenates an identifier to the result, and broadcasts the resulting message to the controllers group. If the controller receives its own message first, it finds out that its replica is the fastest and therefore is the one responsible for sending the reply to the client; otherwise its message is discarded. This algorithm guarantees that only one replica answers to the client request, because all the messages broadcast in the group are observed by each member in the same relative order (a total order imposed by the use of a atomic broadcast protocol).

Finally, the broadcast of a message *end_of_processing*, after sending the results to the client, by the controller of the fastest replica, closes the processing cycle in terms of the client request. This message makes it possible to work out strategies to detect faults in the fastest replica controller and its substitution by another controller for sending results to the client.

The protocol shown above covers up errors due to timing faults. Concerning treatment of failed elements, two detection procedures are foreseen in the original literature. [17]:

- A weak coupling is perceived between the controller and its replica. In this case, *time-out* mechanisms are maintained in the controller to detect the lack of an associated replica ;
- Competitive replication gives a privilege to the fastest replica and, consequently, can lead to considerable asynchronism in the set of replicas. This asynchronism is dealt with, by periodically having a *rendezvous,* in which all the controllers broadcast the results of their replicas among themselves and the last to broadcast is the one which sends the results to the client. This *rendezvous* is limited in time, so as to allow for the detection of failed controllers.

3. Reflective structure for the competitive model

The *computational reflection* paradigm allows a system to execute processing on itself, in order to modify its behavior. In [13], the reflective paradigm is introduced into the object oriented programming using the *meta-objects* approach. Here, the functional and non-functional aspects are separated through the use of *base-objects* and *meta-objects.* A meta-object is associated with each base-object. Through their methods, the base-objects express the application functionalities, while the associated meta-objects carry out control procedures that determine the behavior of their corresponding base-objects. The calls to the base-object methods are trapped, so as to activate the meta-methods that make it possible to modify base-objects behavior or add functionalities to their methods.

In this study, computational reflection is used to develop an integration model for replication techniques in open systems. The reflective paradigm allows us to assign to the base-object the functionalities of a replicated application, while meta-objects execute replica coordination protocols. This model allows the use of different replication techniques while the base-objects maintain their characteristics; to this end, all that is needed is to change the associated meta-objects.

The structure proposed for incorporating active replication concepts into the reflective processing model is presented in figures 2 and 3. Each replica was mapped under the form of an base-object, with which a meta-object, assuming the functionality of controller, is associated. The competitive replication that we use follows a failure semantic of crash. Since we accept a strong coupling between the controller and the associated replica, the errors generated will be attributed to both; in the *crash* failure, the controller and associated replica will cease their execution.

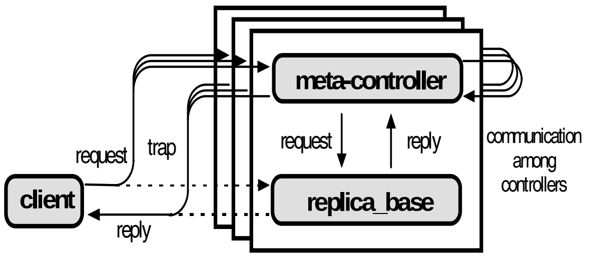

figure 2: Reflective structure for the active competitive replication model.

A request multicast by the client into a group of replicas is trapped to the respective controllers, that, in turn, have to interact in order to implement the coordination protocols of the replication scheme used. The actions of a controller are succinctly described in the code of figure 3. Each base-object method is associated with a meta-method in the controller, responsible for its activation (*method base_1 and meta_method_1,* in the figure cited).

The *meta_control* method implements the coordination protocol among the replicas described in the preceding section. The basic behavior of the algorithm consists of iterating between the choice of a replica for the reply to the client (`first`) and the closing procedure, until there is a confirmation that the reply has actually been sent (`concluded` condition of the `while` loop**).** It is simple to verify the termination of the request processing: if, after multicasting the method `closing`, the fastest replica (`first`) is still alive (into the *membership*), then the reply was actually sent. Otherwise, a new replica is chosen and the process is repeated. This procedure eliminates the need to multicast a message about the end of the processing. In the algorithm, the activations of the methods `multicast_id` and `closing` are transmitted to all the replicas of the group, in a totally ordered manner.

```
class meta_controller {
   // declaration of variables

   method meta_method_1 (parameters) {
      method_base_1 (parameters);
      meta_control  (parameters);
   };

   ... // declaration of further meta-methods

   // implementation of the meta-control method
   method meta_control (parameters) {
      first := null ;
      concluded := false ;
      my_id := get_system_id () ;
      while not concluded do
         if (first = null) then
            group.multicast_id (my_id);
         end ;
         if (first = my_id) then
            // first replica to reply
            return ; // return reply to the client
         else
            if not concluded then
               group.closing () ;
            end ;
         end ;
      end ;
   }

   method multicast_id (int id) {
      if (first = null) then
         first := id ; // id of the fastest replica
      end ;
   }

   method closing () {
      if (first ∈ membership) then
         concluded := true ;
      end ;
      first := null ;
   }
```

figure 3: Competitive replication meta-controller.

Both the competitive replication model and the support utilized give a privilege to the fastest replica, what may cause a lack of synchronism in the slower replicas. The periodical execution of the global *rendezvous* technique, proposed in [17] is not used here, due to its cost implications in the system performance. The solution adopted is based on the property of *virtual synchronism* [5], maintained by the lower layers of the support used in this implementation. In this way, as long as the replica belongs to the *membership* of the group, it will have the same messages in the same order as the others. When the input buffer in the communication support associated with the slowest coordinator/replica pair, reaches the limit of its capacity, the support withdraws the replica from the *membership.* A replica can detect its exclusion and reintegrate itself to the group, through the `view_change` *(view)* method, defined in the interface BOA of Electra and activated automatically by the support for each change in the *membership.* The activation of this method is not preemptive, occurring only after processing the current method. The body of the method `view_change` is defined according to the application characteristics. In this way, in our implementation we carried out a membership test in the body of this method: if the replica has been excluded (`view.number=1`), the BOA function `join` *(group)* is activated, thereby effecting its reintegration into the group.

Regarding the crashes that may occasionally occur in the evolution of the system, our implementation provides procedures for recuperating the degree of replication. If the number of active replicas in the group falls below a preestablished limit, the oldest replica takes the initiative of producing new replicas, in this way, reestablishing the ideal number of replicas. The code referring to these recuperation procedures is based on a *membership* test (`view.number < quorum_minimum`), and it is inserted into the `view_change` method cited above.

Our replica state recovery approach differs from that proposed in [7], in which the recovery occur through meta-methods making updates in public attributes of their associated replicas, with the use of appropriated coordination protocols. In our approach, we utilize more support-provided primitives and fewer coordination protocols, what simplifies the state recoveries. The state recovery, in our system, is based on the primitive *join*, offered by the support, and activated through the `view__change` method.

In object-oriented languages, each meta-object is an instance of a class on the meta-level that defines its structure and behavior. In this article we limit ourselves to talking only about meta-objects because we are interested in emphasizing the aspects of execution time of the meta-objects approach. In [8], these aspects added to other referents to the use of the same approach in real-time applications, are approached within the structure of a language that is being developed.

4. The CORBA support utilized

The implementation of the replication model presented in section 3 presupposes the existence of an run-time support that offers facilities for programming distributed objects. A platform conceived based on the concepts of the CORBA (*Common Object Request Broker Architecture)* standard is to provide the necessary support for distributed object-oriented applications. In this section we briefly describe the Electra system, a CORBA platform utilized in our implementations.

4.1 CORBA architecture

CORBA specifications form a set of standards and concepts proposed for open systems by OMG (Object Management Group) [15]. CORBA architecture is composed of an ORB (Object Request Broker) kernel, that implements communication abstractions among distributed objects and an interface management structure that contains static and dynamic invocation interfaces, object adapters, interface and implementation repositories (figure 4).

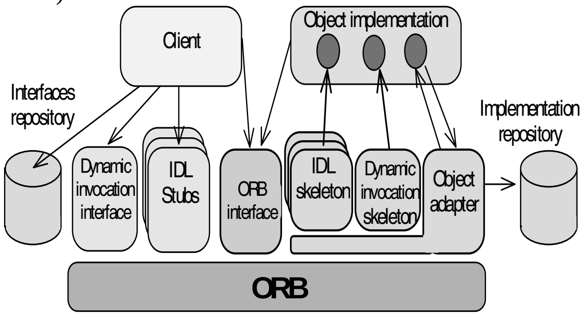

figure 4: The CORBA architecture.

In a CORBA environment, each object has its interface specified through an Interface Description Language (IDL), a declarative language with syntax and predefined types based on the language C++. The interactions follow the client/server model. The CORBA client, in a service request, utilizes stubs generated in the compilation of the IDL specification of the server object, or builds this request, using the dynamic invocation interface DII. To allow for dynamic invocations, object interfaces must be stored in the interface repository. The client's request is transmitted over the network, using the ORB, that transfers the control to the object adapter to activate the operation in the implementation of the server object, by means of the IDL skeletons.

The original CORBA proposal does not provide for adequate support mechanisms to groups of objects. To fill in this gap, some extensions to the CORBA standard have been proposed in terms of incorporating this concept. Electra [14] is a product of these efforts.

4.2 Electra

Electra [14] is an *Object Request Broker* (ORB), compatible with the CORBA standard [15], presenting support to object groups. This platform combines the benefits of the CORBA standard with the power of lower level tools for group processing, such as Horus [18], Isis [5], and others. Interactions in Electra can occur as reliable multicast or point-to-point communications. Ordering mechanisms (total, causal and fifo) are offered to guarantee consistency among members of the object group. The client makes use of a given method invocation model, regardless of whether the server is a single object or a group. These invocations may be synchronous, asynchronous (*one-way*) or semi-synchronous (*deferred-synchronously*), through static or dynamic interfaces. On multicasting a method call, through a CORBA static or dynamic invocation interface, the programmer has at his disposal two modes of group communication in Electra:

- Transparent: the group is seen as a simple and completely available object, and the client only receives the final result furnished by the group;
- Non-transparent: permits access, in an invocation, to the results of each individual member of the group of objects.

In the interface BOA (*Basic Object Adapter)* of Electra, services referring to group management are added, such as *creating* a group of objects, *including* objects in the group or *excluding* them from it, *selecting* a protocol of *multicast*, *membership* and transfer of state, and so on. These services are provided by the lower level tools used, such as Horus or Isis.

5. The integration model of replication techniques in open systems

In the previous section, we could see that the CORBA/Electra platform offers adequate support for group processing. In this section, we describe the integration of the reflective model proposed in section 3 over the Electra system.

5.1 The integration model in CORBA platforms

Figure 5 explicits the integration model of replication techniques within the CORBA context. The access to the support provided by a CORBA platform is available both to the server and to the client by entities represented as *meta-objects (client and server)* and identified generically as *meta-communication.* These entities are actually nothing more than the set of *stubs* for the client and server, *stubs* for the communication among the replicas and the BOA interfaces providing the group management. All these *stubs* are generated by the translation of the IDL [16] declaration of a server object. The use of the term "abstract object" given to meta-communication on the model follows some authors [9] and has the sense of a simple separation for greater clarity. In reality, these interfaces are generated in Electra as a set of methods that will be composed of multiple inheritance in the client and controller meta-objects (section 5.2).

In the model the client introduces itself within a *client-base* structure, that represents the application behavior, and a *meta-client*, that does not present an active function in our implementation, but that could be used in managing the replicated client, or to implement mechanisms for handling exceptions in the client. The structure of each server replica is similar to that of the client: a *replica-base* object, carrying out the replicated service, and a *meta-controller*, responsible for executing the coordination protocol of the replication, like one described in figure 3.

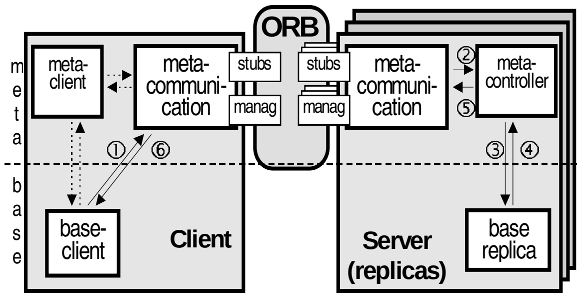

figure 5: Structure of the model on a CORBA support.

The numbered arrows in figure 5 indicate the normal way of a client request: The request made by the client base (1) is then broadcast using a stub appropriated in the client meta-communication. In each replica, the meta-communication, by means of a local stub, receives the request and transfers it to the meta-controller (2), which then activates the local replica (3). On receiving the reply (4), the meta-server executes the coordination protocol, by means of the meta-communication so as to interact with other replicas. The processing and interactions on the level of the meta-controllers are conditioned at this time by the replication model utilized. Later, the reply is then sent back to the client (5 and 6).

This model can be used in other replication techniques, the differences centering mainly in server replicated meta-controllers. In some techniques the meta-communication entities may gain functionality, besides that of concentrating methods of access to CORBA supporting services. For example, in the use of active replicas with voter and adjuster mechanisms, the implementation of voting or adjustment can be programmed on the client's side in a more simplified form. Transparency could be achieved in this case, implementing these mechanisms in the client meta-communication entity, which, with the addition of this functionality, takes on the characteristics of a real object.

5.2 Building replicated services following the integration model

The first step in the building of a system on a CORBA/Electra platform is a description in IDL of the meta-controller interface, following the specification of the replicated service provided by the server to the client. Besides this interface, due to some limitations imposed by the Electra, it is necessary to declare a second one, for implementing the replica coordination, composed by methods that provide communications among replicas.

```
// IDL

interface meta_controller_1
{
   // Description of the data types employed

   // Description of the server methods
   boolean meta_method_1 (parameters);
   ...
   boolean meta_method_n (parameters);
};

interface meta_controlller_2
{
   // Description of the meta_controller methods
   boolean broadcast_id (in int id);
   boolean closing ();
};
```

figure 6: IDL interface of the replicated server.

Figure 6 presents both IDL declarations of a replicated server in according to the specifications described in figure 3. The interface *meta_controller_1* allows clients access to the services offered, while the interface *meta controller_2* declares the methods necessary for intra-replica interactions. It should be pointed out that both interfaces are actually two facets of the same server (or, in our case, of the same group of objects).

In compile-time, the Electra/IDL compiler automatically generates the whole support for communication (*stubs)* among the entities involved, including, as well, the functionalities for group management of the BOA (in Electra, every object is an instance of a sub-class of the BOA class). The compiler also generates files containing structures (declarations of variables and methods) for including the client and server codes. The programmer, then, is responsible for the implementation of the replicas (base-objects) and the replica coordination suitable (meta-objects), by filling the bodies of the methods declared in the interfaces. With this implementation scheme, illustrated in figure 7, the client and server base-objects are kept devoid of all activities that are not related to the application itself. All aspects related to the replica coordination and the interactions in the CORBA context, are concentrated at the meta-level.

Our implementation was carried on a UNIX platform, where each associated pair base-object/meta-object was intended to share the same process, making the interactions between them local, without the need for ORB. The needs for concurrence between base-objects and meta-objects within a process are satisfied by the use of a threads library offered by the Electra support. However, the current version, (1.0) of this platform does not support pre-emptive threads, which limits the degree of concurrence in dealing with client requests. As a result of this restriction, it becomes difficult to implement replication techniques, which has forced us to seek alternative implementation solutions. The solution adopted consists of separating the functionalities of the meta-controller into two UNIX processes.

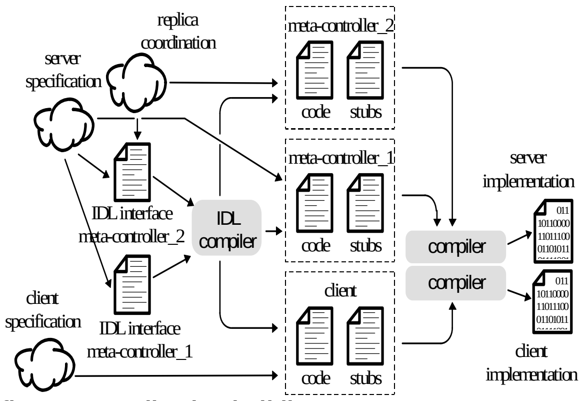

figure 7: Application building process.

Due to the fact that the language used (C++) has no specific constructions to support reflection. The reflection is implemented artificially, through the direct activation to the meta-method, in the client code. The use of a language supporting reflection, as is the case with *Open C++* [6], might eliminate this problem, but in this case, the IDL compiler of the CORBA environment used should support this language.

6. Considerations about the results

Redundancies and fault tolerance implementations can follow several approaches [7]. The implementation of fault tolerance techniques by means of runtime support offers on the application level some degree of transparency concerning the coordination aspects of the technique used. The disadvantage is that once the fault assumptions and the replication technique are chosen in configuration time, we will have defined a specific execution support. The approaches of library and languages for the implementation bring aspects of coordination to the programmer level, without, however, separating them from functional application aspects.

Computational reflection permits independence of the replica codes in relation to the coordination protocols, leading to a greater flexibility in the system: changing the technique or altering it by meeting desired degrees of fault tolerance, may simply result in switching the coordination protocols on the meta level, involving no alteration in application algorithms or in the run-time support what is suitable in open systems. The use of the reflective computing for implementing fault tolerance techniques is proposed in [3] [7], and the separation between the coordination and the replicas has already been recommended in [17].

The model presented was used for implementing the active competitive replication protocol, described in section 2. The implementation carried out makes intensive use of the Electra support facilities, which makes easier the coordination needs of the technique implemented. Furthermore, the uses of a CORBA platform has allowed the implementation of our application on an heterogeneous system (local network of machines running SunOS 4.X and Solaris), facilitating aspects of interoperability.

The integration structure proposed has proved to be quite flexible, other replication techniques can be easily implemented. Up to now, we have implemented the *primary/secondary* , *leader/followers* and *cyclic redundancy* techniques, using the same integration model. The necessary changes for the substitution of replication techniques in the integration model are limited to the IDL meta-controller interface and their codes that implement the coordination protocols.

These replication models were applied in the implementation of a multimedia application (animation viewer accepting the *MPEG* format). Simulations of *crashes* were carried out, utilizing these implementations. In all these replication techniques utilized, the continuity of service was obtained in case of failures, since the premises of each technique were respected. At present, we are working out detailed measurements on the performance of the replication techniques cited using the Electra platform. We are also porting our work to the Orbix+Isis platform [10].

7. Conclusion

An integration model for replication techniques in open distributed systems was presented in this article. The use of computational reflection concepts makes it possible to obtain the necessary flexibility for developing and implementing different replication models for fault tolerance in these environments.

Within this context, the work presented in this article continues at present in various directions. The validation of the model proposed through application in real situations and the incorporation of language constructions in terms of facilitating the programming of the reflective model are some of the current activities involving this work.

The programming model presented in this article is part of a cooperative research project, sponsored by the brazilian state agency CNPq (PROTEM-CC project), and has as its aim to build an object-oriented environment that supports distributed applications with requirements of real-time and fault tolerance.

Acknowledgment

We would thank S. Maffeis, author of the Electra system [14], for his kind attention helping us to solve our main difficulties in the beginning of this work.

References

[1] E. Oskiewicz, N. Edwards, **“A Model for Interface Groups”**, ANSA Phase III technical report APM.1002.01, Cambridge-UK, may 1994.

[2] R. M. Adler, **“Group-Oriented Coordination Extensions to OMG´s OMA/CORBA”**, OMG Presentation, San Jose - CA, June 1995.

[3] G. Agha, S. Frolund, R. Panwar, D. Sturman, **“A Linguistic Framework for Dynamic Composition of Dependability Protocols”**, Proceedings of the DCCA-3, 1993.

[4] K. Birman, T. Joseph, F. Schmuck**, “ISIS - A Distributed Programming Users Guide and Reference Manual”**, The ISIS Project, Department of Computer Science, Cornell University, Ithaca - NY, march 1988.

[5] K. P. Birman, **"The Process Group Approach to Reliable Distributed Computing"**, Technical Report TR 91-1216, Cornell University Computer Science Department, Ithaca, N.Y., July 1991.

[6] S. Chiba, **“Open C++ Programmer’s Guide”**, Technical Report 93-3, Department of Information Science, University of Tokio, 1993.

[7] J. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud and Z. Wu, **“Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming”,** Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing, Pasadena (CA), June 1995.

[8] J. Fraga, J.-M. Farines, O. Furtado, F. Siqueira, “**A programming model for real-time applications in open distributed systems**”. Proc. of the 2nd IEEE Workshop on Future Trends in Distributed Computing Systems, august 1995.

[9] O. Hagsand, H. Herzog, K.P. Birman, R. Cooper, **“Object-Oriented Reliable Distributed Computing”**, 2nd IEEE International Workshop on Object-Orientation in Operational Systems, 1992.

[10] Isis Distributed Systems Inc., IONA Technologies, Ltd. **"Orbix+Isis Programmer’s Guide"**, 1995. Document D070-00.

[11] Isis Distributed Systems Inc., **"RDO/C++ Tutorial Release 1.0.3"**, Apr. 1994.

[12] M. C. Little, **“Object Replication in a Distributed System”**, PhD. Thesis, University of Newcastle upon Tyne Computing Laboratory, September 1991.

[13] P. Maes, **“Concepts and Experiments in Computational Reflection”**, OOPSLA 87 Proceedings, pp. 147-156, October 1987.

[14] S. Maffeis, **"Adding Group Communication and Fault-Tolerance to CORBA"**, In Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies, Monterey - CA, June 1995.

[15] Object Management Group, **"The Common Object Request Broker: Architecture and Specification"**, Revision 1.2, OMG Document, December 1993.

[16] Object Management Group**, “IDL C++ Language Mapping Specification”**, OMG Document 94-9-14, 1994.

[17] D. Powell, **“Delta-4 Architecture Guide”**, Esprit II P2252, Delta-4 Phase 3, August 1991.

[18] Robbert V. Renesse and Kenneth P. Birman, **"Protocol Composition in Horus"** Dept. of Computer Science of the Cornell University, Mar 1995.

[19] F. B. Schneider, **“Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial”,** ACM Computing Survey, 22(4):299-319, Dec 1990.