Fault Tolerance as an aspect using JReplica[1]

José Luis Herrero[1], Fernando Sánchez[1], Miguel Toro[2]
[1] Computer Science Department
University of Extremadura.Spain
{jherrero, fernando}@unex.es
[2] Computer Science Department
University of Sevilla.Spain
mtoro@lsi.us.es

Abstract

Reliability and availability are very important trends in the development process of distributed systems. In order to improve these features, object replication mechanisms have been introduced. Programming replication policies for a given application is not an easy task, and this is the reason why transparency for the programmer has been one of the most important properties offered by all replication models. However, this transparency for the programmer is not always desirable. In this paper we present a replication model, JReplica, based on Aspect Oriented Programming (AOP). JReplica allows the separated specification of the replication code from the functional behaviour of objects, providing not only a high degree of transparency, as done by previous models, but also the possibility for programmers to introduce new behaviour to specify different fault tolerance requirements. Moreover, the replication aspect has been introduced at design time, and in this way, UML has been extended in order to consider replication issues separately when designing fault tolerance systems.

1: Introduction

This work tries to introduce replication in object orientation by means of a new aspect. For this purpose, a new language called JReplica has been developed. This language captures the relevant aspects of replication, and encapsulates them into a component or group of related components favouring the reusability and dynamic adaptability of replication policies. This language also favours the use and reuse of replication policies independently from the middleware used to communicate objects (currently several implementations of CORBA and RMI).

The work is not limited to the definition of this new language. AOP ideas have been translated to the design level. In this way, the semantic of UML has been extended in order to represent replication properties. From a given design, the same for whatever middleware, a visual tool is able to generate code.

The rest of the paper is as follows: section 2 explains the different approaches to introduce replication in object orientation. Our proposal is introduced in section 3. Section 4 shows related works. Finally, future works are outlined in section 5.

2: Fault tolerance approaches

There are several different approaches to introduce fault tolerance in object oriented systems. They can be categorised in the following way:

1. **Integration approach:** In this approach, replication is integrated inside the model. Replication is coded inside

[1] This work has been developed with the support of CICYT under contract TIC99-1083-C02-02

the ORB so each ORB must be modified in order to provide fault tolerance. Electra [1], Orbix+Isis [2] are two models that are based on this approach (figure 1).

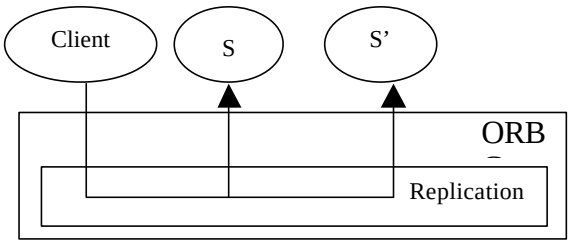

Figure 1. Integration approach

2. **Interception approach:** In this model, every message is intercepted and redirected to a replication toolkit. This new tool is in charge of providing fault tolerance. The ORB must be modified introducing the interception mechanism. Eternal [3] is an example of this approach (figure 2).

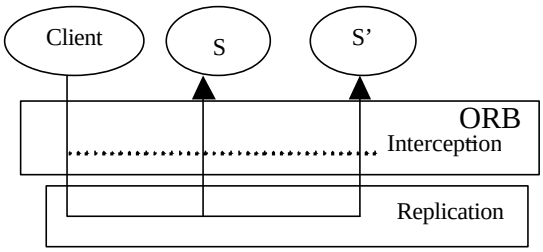

Figure 2. Interception Approach

3. **Service approach:** A new replication service is added to the ORB. This service provides mechanisms for object replication. OGS [4] and the new Corba Fault Tolerance specification [5] are based on this approach (figure 3).

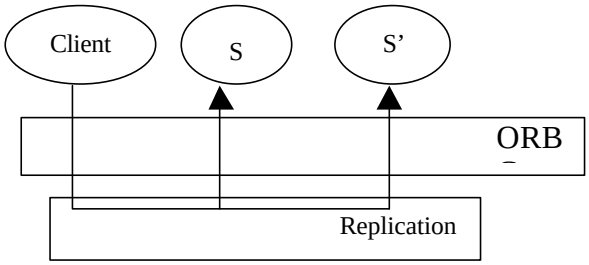

Figure 3. Service Approach

All these models introduce new elements to provide fault tolerance through replication. Transparency is the most important property achieved. In this way, programmers do not have to take care about replication, and they do not need to define any protocol to develop fault tolerance applications because replication is obtained automatically by the model. However, all these models have two main drawbacks in the following sense:

- **Close:** A totally transparent system doesn't allow programmers to change replication mechanisms. Replication properties can not be established, such as the replication granularity, or the moment when replication protocols must be executed. These properties are defined automatically by the model and they are the same for every system. In this way, programmers can not take advantage from system requirements.

- **ORB dependent:** Replication depends on the ORB implementation. Any replication policy must be coded into an individual ORB, and it can not be reused in a different ORB. There's no way to port the same replication policy to other ORBs.

Although transparency is a good property to be achieved, it is not always necessary, moreover, sometimes it is not desirable. Sometimes the nature of the problem may require establishing the replication properties and behaviour by the programmer. Even more, if requirements guide the replication behaviour, the system could take advantage of them, and system performance could be increased. If the replication model is totally transparent, there is no way to define fault tolerance applications according with system requirements.

3: Proposal

The model here proposed is based on the paradigm of Aspect Oriented Programming (AOP). Our research group has gained experience with AOP during the last few years working with the synchronization, coordination and distribution aspects [6, 7]. Here we go one step further introducing the replication aspect as a new non-functional property of the object. With this, separation transparency is granted because replication policies can be reused among applications with no changes. In addition, programmers can get control over the replication policy using the specific replication language provided: JReplica.

3.1: Framework

The proposed model keeps aspects separated from the functional code of the object. A reflexive architecture is used in order to introduce two different levels of execution:

- **Functional Level:** Object functionality is defined at this level. Two new entities (in, out) have been attached to each object in order to communicate objects with its aspects.

- **Aspect Level:** Aspects are defined at this level. Each object can be associated with one or more aspects.

The model is show in the figure 4.

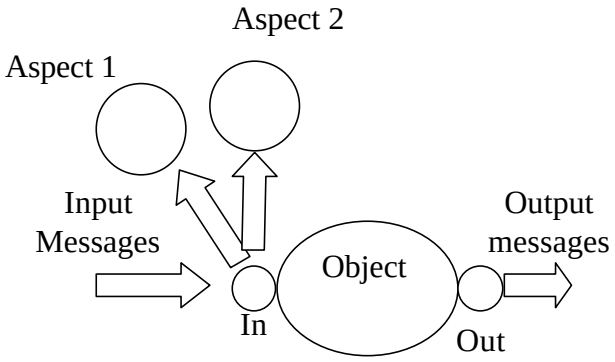

Figure 4. The aspect model

Each object is composed of three different entities:

- **In:** This entity intercepts all the input messages and redirects them to the aspect level.

- **Functional Object:** This is the place where the basic behaviour of the object is implemented.

- **Out:** This entity intercepts all the output messages and transforms them into the right middleware (CORBA or JavaRMI).

3.2: Replication Aspect

According to figure 4, this work tries to focus on the replication aspect (figure 5). Passive replication is the replication technique that has been considered. Although there are other different replication techniques such as active replication, passive replication allows replicating deterministic and non-deterministic objects, while active can not.

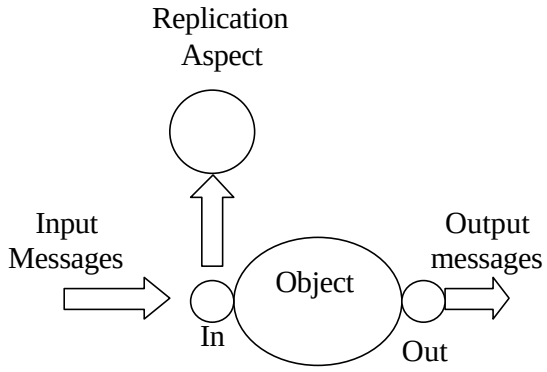

Figure 5. The Replication Aspect

Under this previous consideration, the replication aspect must perform the following tasks:

- **Decide when to activate the replication mechanisms:** It's very important to define when to send replication messages because they will affect the performance of the whole system. If these messages are sent very often, the system will collapse. While if not, copies can be inconsistent for a long time. There are two different ways to establish this activation:

  - Direct: replication is activated just after a method of the object has been executed.

  - Indirect: replication is activated depending on the value of some conditions that must be checked periodically.

- **State Update:** When the replication mechanisms are activated, the replication aspect captures the state of the object and sends it to every copy. In order not to break object encapsulation, get and set methods are called to obtain and modify the state.

- **Fault Checking:** In order to test faults, a pin protocol is developed. This protocol ensures that if a fault happens, each copy will be notified.

- **Recovery:** When a fault is found, another protocol will select one of the replicas to take the control.

Figure 6 shows a representation of a fault tolerance system using this aspect model.

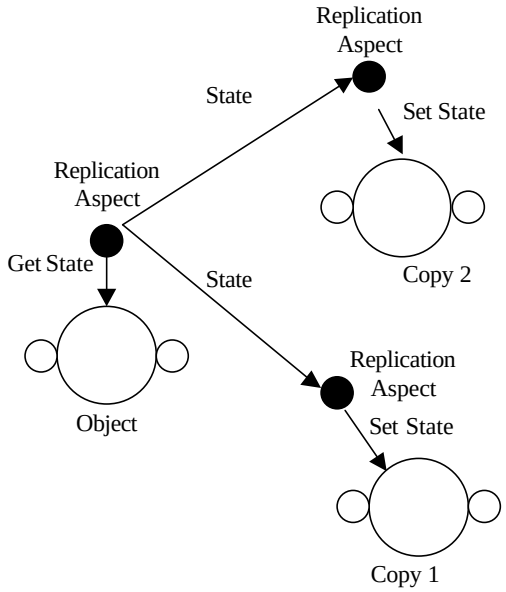

Figure 6. Aspect system example

The main advantages of this model are the followings:

- Those benefits derived from the **use of AOP**, mainly modularity, reusability of code and adaptability of applications.

- **ORB Independence:** Replication algorithms are independent from de ORB. In a previous work [7] different distribution protocols were defined as a separated aspect providing a dynamic, adaptable and transparent object distribution. Now, as the replication module is defined outside the ORB, the combination of distribution and replication aspects offer the possibility of reusing the same replication policy in different ORBs.

- **Open**: Thought replication algorithms are hidden and separated from object behaviour, replication properties and behaviour can be defined. Reflective mechanisms can communicate the object level with the replication level. This communication provides the way to introduce new replication actions.

3.3: JReplica: Java Fault Tolerance Language

JReplica is a language with the only purpose of defining replication policies. Its syntax is based on Java. It introduces new primitives, which are shown in figure 7. This Java extension introduces two main elements:

1. **Replication Policy:** A new entity called *Disguise Replication* defines the replication aspect. This entity is divided into the following parts:

   - **Attributes:** the information that defines the replication policy.

   - **State:** the set of replication states.

   - **Operations:** methods that can manipulate the replication state.

   - **Guard:** a condition that must be true before replication. If this condition is false, replication won't be executed.

   - **Before Replication:** the set of actions that must be executed just before replication.

   - **After Replication:** the set of actions that must be executed just after the replication is executed.

   - **Error:** the set of actions that must be executed when a replication error appears.

2. **Composition:** A class can be composed with different aspects. In our case, this means that every object will extend its functionality with replication mechanisms.

3.4: Representing Replication at Design Level

Replication policies now can be defined with the JReplica language. This language helps programmers to define easily replication properties in object oriented systems. But we consider that replication must be

introduced at earlier stages of object life cycle, more concretely at design level. In this way, UML [8] is used as the modelling language due to it being a standard. As UML does not provide mechanisms to represent replication, its semantic has been extended in order to express replication properties and behaviour.

```
Class  <name>
{ ........
}

.....................
x=new C1
Compose x with R;
y=new Replica of x;
.....................
```

```
Diguise Replication <name>
{
  Attributes: .....
  Operations: .....
  State:  .....
  Guard: .....
  Before Replication: ....
  After Replication: .....
  Error: .....
}
```

Figure 7. JReplica replication primitives

UML semantic can be extended with the introduction of new stereotypes. At this point, we have considered that replication policies can be designed separately and independently, in the same way as has been explained at the implementation level. As such, the aspect concept is introduced in UML to express the AOP philosophy. The replication aspect is represented with a new stereotype, called <Replication>. This new stereotype is shown in figure 8. The replication stereotype represents a particular replication policy. Information is represented as follows:

- **Stereotype Attributes:** Represent the information that defines the replication policy.
- **Stereotype Methods:** Define the set of methods that can manipulate the replication state.

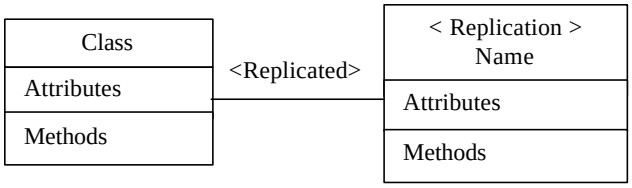

Figure 8. UML extension

As it can be shown, there are other elements that can not be represented in this stereotype. The dynamic behaviour of replication can not be represented in a normal class diagram. Statechart diagrams represent dynamic behaviour. So the solution goes by attaching a statechart diagram to this replication stereotype. In this way, replication static properties and dynamic behaviour can be designed. The dynamic behaviour of replication policies can be represented in a statechart diagram as it is shown in figure 9.

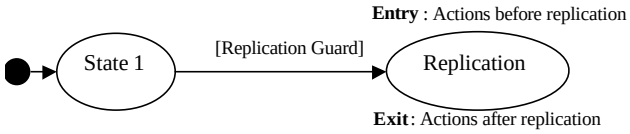

Figure 9. Statechart Representation

The elements that are represented in this statechart diagram are:

- **State:** Each replication state is represented by an state. There is a special state called *Replication* that represents the moment when replication is to be executed.
- **Guard:** Guards are represented in the transition of each state.
- **Before Replication:** The set of actions that is executed just before the replication begins is represented in the entry actions of the Replication state.
- **After Replication:** The set of actions that are executed just after the replication ends are represented in the exit actions of the Replication state.
- **Error:** Replication errors are represented as a new state.

A tool is being developed in order to generate JReplica code starting from this extension of UML. In this way Replication aspect has been introduced from design to implementation level. This tool is based on other one we have developed for the synchronisation aspect [9].

Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)
1071-0485/01 $17.00 © 2001 IEEE

4: Related works

There are several models that provide replication mechanism to achieve fault tolerance. In [10], a new interception mechanism called Aroma is introduced in the Java RMI architecture. Other models are based on the introduction of separated entities that implement replication protocols. The Cadmium Model [11] defines a couple of new entities called *Stub* and *Scion*, which are attached to a client and a server respectively and offer replication mechanisms. In [12] a new replication entity and a consistency manager are introduced, both separated from the object. In AspectIX [13] a single object is divided into fragments, all of which have a different purpose. One of these fragments offers replication facilities. The GARF [14] model defines two different entities in order to introduce replication, they are called e*ncapsulator* and *mailer*. A two level reflective architecture was defined for Java in [15]; object functionality is defined in the first level, while replication protocols are established in the second one. All these models only take into account the implementation level, they are focused on replication protocols and the definition of a framework that provides fault tolerance, ignoring the design phase.

A new pattern [16] has been defined in order to provide support for the representation of replicated objects. Moreover, a new language that helps programmers to build fault tolerance systems has been defined in [17, 18]. This proposal is based on the concept of separation of concerns and extends AspecJ language [19] with replication primitives. It is possible to define the attributes that need replication and what to do when a replication error happens. But there is no way to express new replication actions or when replication must be executed. Although these models help programmers to implement fault tolerance systems, it is necessary to introduce mechanisms that help software engineers to design this kind of requirements.

5: Future works

Future works will consider extensions to JReplica in order to express more complex replication mechanisms. The current version showed us the suitability of the model.

References

[1] S.Maffeis. *Run-Time support for object-oriented distributed programming*. Phd Thesis, University of Zurich, 1995.

[2] IONA and Isis. *An Introduction to Orbix+Isis.* IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[3] L.E.Moser, P.M. Meliar-Smith and P. Narasimhan. *Consistent object replication in the Eternal system.* Theory and Practice of Object Systems, 81-92, 1998.

[4] Pascal Felber. *The CORBA Object Group Sevice. A Service approach to object groups in CORBA.* Phd Thesis 1998. University of Lausanne.

[5] OMG TC document ptc/2000-03-04. *Fault Tolerant CORBA.* Draf Adopted Specification. 2000.

[6] J.M. Murillo, J. Hernández, F. Sánchez, L.A. Álvarez. *Coordinated Roles: Promoting Reusability of Coordinated Active Objects Using Events Notification Protocols.* In Coordination Languages and Models. Springer-Verlag, LNCS 1594, April, 1999.

[7] F. Sánchez, J.Hernández, J.M.Murillo, J.L.Herrero, R.Rodríguez. *Adaptability of Object Distribution Protocols Using the Disguises Model Approach.* 2nd Intl. Symposium, Distributed Objects & Applications (DOA 2000).

[8] Object Management Group. *Unified Modeling Language, version 1.3.*

[9] J.L.Herrero. *Introducing separation of concerns at design time*. PhDOOS Workshop, European Conference on Object-Oriented Programming (ECOOP'2000).

[10] N. Narasimhan, L.E. Moser and P. M. Melliar-Smith. *Transparent Consistent Replication of Java RMI Objects.*2nd Intl. Symposium, Distributed Objects & Applications (DOA 2000).

[11] Aline Baggio. *Adaptable and Mobile-Aware Distributed Objects*. PhD Thesis, Université Pierre et Marie Curie and INRIA, Paris, France, June 1999.

[12] Georges Brun-Cottan and Mesaac Makpangou. *Adaptable Replicated Objects in Distributed Environments.* BROADCAST TR No. 100. Appeared in the proceedings of the 2nd BROADCAST Open Workshop, Grenoble, July 1995.

[13] Martin Geier, Martin Steckermeier, Ulrich Becker, Franz J. Hauck, Erich Meier, Uwe Rastofer. *Support for mobility nd replication in the AspectIX architecture.* Object-Oriented Technology, ECOOP'98 Workshop Reader, LNCS 1543, Springer, 1998; pp. 325-326.

[14] B. Garbinato, R. Guerraoui, and K. R. Mazouni. *Implementation of the GARF replicated object platform.* Distributed Systems Engineering Journal, 2:14-27, 1995.

[15] Jürgen Kleinöder, Michael Golm. *Transparent and Adaptable Object Replication Using a Reflective Java.* Tech. Report TR-I4-96-07, Universität Erlangen-Nürnberg: IMMD IV, Sept. 1996.

[16] Teresa Gonçalves and António Rito Silva. Passive Replicator: *A Design Pattern for Object Replication.* Second European Conference on Pattern Languages of Programs. July 1997.

[17] Johan Fabry. *Replication as an Aspect - The Naming Problem.* ECOOP Workshops 1998: 424-425.

[18] Johan Fabry. *A Framework for replication of objects using Aspect-Oriented Programming.* Phd Thesis 1998. University of Brussel.

[19] C.V. Lopes. *D: A Language Framework for Distributed Programming*. Phd Thesis 1997. University of Northeastern.

Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'01)
1071-0485/01 $17.00 © 2001 **IEEE**