CORBA Replication Support for Fault-Tolerance in a Partitionable Distributed System*

Technical Report ITI-ITE-06/01

Stefan Beyer and Francesc D. Muñoz-Escoí and Pablo Galdámez
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
Spain
{stefan, fmunyoz, pgaldamez}@iti.upv.es

Abstract

The Common Request Broker Architecture (CORBA) specification originally did not include any support for fault-tolerance. The Fault-Tolerant CORBA standard was added to address this issue. One drawback of the standard is that it does not cover fault-tolerance in the case of network partitioning faults. However, wide area networks, over which distributed systems are often employed, are especially susceptible to network partitioning.

The main contribution of this paper is the design of a fault-tolerance CORBA add-on for partitionable environments. In contrast to other solutions, our modular design separates replication and reconciliation policies from the basic replication mechanisms. This modularity allows the replication and reconciliation strategies to be modified easily.

1 Introduction

The Common Request Broker Architecture (CORBA) [Obj04a] is a popular middleware framework to construct distributed object systems. As distributed systems are subject to host and network failures, fault-tolerance is an important aspect in the design of such systems. However, the CORBA specification did originally not include any support for fault-tolerance. Since then, the Fault-Tolerant CORBA specification (FT-CORBA) [Obj04c] has been added to introduce a degree of fault-tolerance to CORBA. However, the standard has various drawbacks. One important shortcoming of FT-CORBA is that it does not provide support for fault-tolerance in a partitioned network. Wide area networks, over which distributed systems are often employed, are especially susceptible to network partitioning.

In this paper we present the architecture of a middleware add-on that adds fault-tolerance to CORBA in a partitioned environment by means of replication. The system is part of the DeDiSys project [OFG+06]. DeDiSys aims at providing fault-tolerance through add-ons for various middlewares. The CORBA add-on presented here uses CORBA Portable Interceptors [Obj04b] to intercept calls to server objects in a transparent manner and divert these calls through a replication manager. An underlying group membership and communication service provides reliable communication.

In contrast to other systems, the modular design of the DeDiSys replication support allows different replication and reconciliation policies to be implemented easily. The design of the replication support is based on a separation of mechanism and policy. Replication mechanisms are basic primitives such as creating a replica and changing its role, or the ability of managing object and replica references. In our system these mechanisms are provided by a distributed replication manager. Many replication protocols

*This work has been funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract number 004152).

will have these mechanisms in common. In contrast, policies, such as the update propagationand reconciliation policies, may vary between replication protocols. We extract such policy from the replication manager and place it into a replication protocol component. The replication manager and the replication protocol components provide fixed interfaces. New replication protocols can be implemented by replacing the replication protocol component.

A default replication protocol [BBGME05] is included. The protocol allows operations in each partition in a partitioned system to continue. Resulting conflicts can be resolved automatically by the reconciliation support or manually by the application.

A non-CORBA prototype [BSMEG06] of our architecture has been implemented and we are currently in the process of implementing the full CORBA system, taking into account the lessons learnt from the prototype.

2 Related Work

In order to add fault-tolerance to CORBA, certain mechanisms, such as replication, are required. Existing systems either implement the FT-CORBA [Obj04c] standard to provide fault-tolerance or suggest their own fault tolerance extensions. There are two possible reasons for a system not to comply with the FT-CORBA standard. Some systems reviewed here were simply developed before the standard was defined. Other systems try to overcome some of the drawbacks associated with FT-CORBA. As DeDiSys is a research project, aimed at partitionable distributed systems, which are not covered by the FT-CORBA standard, we do not consider FT-CORBA compliance as the main factor for this review.

In literature, approaches to add fault-tolerance mechanisms to CORBA are typically classified into three categories: In the **integration approach**, the ORB itself is modified to include the required fault tolerance mechanisms. It is easy to provide transparency using this approach, but existing commercial ORBs cannot be used. Orbix+Isis [ION94], Electra [LM97] and Maestro [VB98] are examples of systems using the integration approach. More recently, the authors of [LNYY03] and [ZMMS04] have proposed the integration of group communication support by modifying the CORBA Open Communication Interface (OCI) and using the Pluggable Protocols Framework [KOS+99]respectively.

In the **service approach**, the mechanisms required to provide fault tolerance are provided as CORBA services. This approach has the advantage that existing ORBs can be used. However, transparency is difficult to achieve with this approach, as applications have to be aware of the fault tolerance services. Object Group Services (OGS) [FGG96] and Newtop Object Group Service [MSEL99] provide services for object group support which can be used to provide fault-tolerance. FTS [FH02], OPEN EDEN [GHN03], IRL [BM03] and AQuA [RBC+03] are examples of reliable CORBA systems using the service approach, although it can be argued that these systems also use elements of the interceptor approach.

In the **interceptor approach**, CORBA invocations are intercepted and redirected to fault tolerance mechanisms. Recent systems make use of CORBA Portable Interceptors [Obj04b]. The only systems using a pure interception approach we are aware of are Eternal [MMSN98] [NMM97] and DAISY [BBC+04].

Three of the systems mentioned above - Maestro, FTS and Eternal - provide some support for network partitioning. Therefore, these systems are reviewed here in more detail. Newtop also provides support for network partitioning, but, as a mere object group toolkit, does not provide any support for reconciling replica state after partitioning. Therefore, we do not discuss Newtop in detail here.

Maestro uses the integration approach. The system was developed before the FT-CORBA specification existed. It is not a pure CORBA implementation, but was designed as a distributed object layer to be used on its own or to be integrated in CORBA or in other distributed object technologies. The system uses Ensemble [vRBH+98] as an underlying group communication and membership toolkit. Partitioning is supported using a variation of the primary partition model [RSB93]. Only updates in one partition are permanently accepted, but in contrast to the regular primary partition model, the decision on which partition dominates is postponed until recovery time. At recovery time the partition with "the most updated" state is chosen.

FTS is an attempt to remain close to the FT-CORBA specification, whilst also providing support for partitioning. The system uses a mixture of the service and interceptor approaches. A group object adapter (GOA) is provided as a CORBA object adapter. The GOA is implemented on top of the portable object adapter (POA) to allow for object groups; that is, groups of replicas representing the same logical object. The main drawback of FTS is that it only implements active replication, although the authors

claim it would be easy to adapt FTS to passive replication. In DeDiSys we use also use the idea of an object adapter providing object group support. FTS uses the primary partition model for consistency in case of network partitioning.

Eternal is probably the most advanced of the systems of which we are aware in terms of support for partitioning, despite being one of the oldest systems. The system allows for active and passive replication. The Eternal replication manager keeps track of replicated objects. CORBA messages are intercepted at the transport level and are redirected using the Totem group communication toolkit [MMSA+96]. Totem provides Eternal with the extended virtual synchrony model, which allows for network partitioning. As far as we know, Eternal is unique in partition-aware CORBA systems, in that it does not use a variant of the primary partition model, but does allow operations in all partitions to continue. A simple reconciliation algorithm is provided. When the network partitions, a primary subgroup is chosen for each object. However, operations are also allowed to continue in secondary subgroups. When subgroups are re-merged, Eternal gives preference to the state contained in the primary subgroup. However, operations in secondary subgroups are queued and applied after the state of the primary subgroup has been installed in all the merging subgroups during recovery. Conflicts that cannot be resolved are reported to the application.

In DeDiSys we make use of some techniques from Eternal, DAISY and FTS. In particular, we use interception, as in Eternal and DAISY, and the implementation of the replication support as a CORBA object adapter, as in FTS. In contrast to Eternal's interception at the operating system level approach we use DAISY's approach of using portable interceptors, which were not available when Eternal was designed. Furthermore, in DeDisys we aim at making replication and recovery flexible and configurable. To this end we do not embed replication protocol and reconciliation policy in the replication manager, as done in Eternal, but provide an easily interchangeable replication and reconciliation protocol component.

3 Design Principles

In order to design a replication support for partionable environments in CORBA we have followed the following design principles:

Separation of Mechanism and Policy Our design distinguishes between replication mechanisms and policy. Replication mechanisms are basic primitives, such as the ability to create a replica or manage the relation between object references and replica references. The provided mechanisms can be used in different ways to implement replication policies, such as the object state transfer policy or the reconciliation strategy. Policies may vary, whereas mechanisms are provided to support different policies. In conventional systems policies and mechanism are often embedded in the same component. This makes it difficult to implement different policies. In DeDiSys, we extract replication and reconciliation policy from the main replication component, which only provides mechanisms that allow to implement a variety of policies.

Interception The DeDiSys concept is to provide a Middleware add-on rather than modifying existing middleware. To achieve this in CORBA we intercept object invocations. To this end, CORBA portable interceptors are used to pass control to the replication support. Interceptors are used on both the client side and on the server side.

Client-Side Transparency Replication should be transparent to the client application. That is, the client is not aware it is dealing with a replicated object and existing CORBA clients do not have to be modified to use DeDiSys, apart from calling a simple initialisation routine.

Server-Side Transparency It is our goal to make server side integration of the replication support as transparent as possible. However, the server application should have some control over the replication support. Therefore, a simple interface provides mechanisms, such as replica creation, and has to be used by the server application. Furthermore, server objects need to implement a simple interface that allows the replication support to access their state. It is our goal to make CORBA server applications as easy to port to DeDiSys as possible, whilst allowing configurability of key parameters, such as number and location of replicas.

4 The DeDiSys Replication Model

DeDiSys aims to introduce fault-tolerance through replication. In this section we describe the failure model we support and the replication model used to achieve this.

The "crash model" [Cri91] is assumed for node failures, and the "link failure model" [Sch93] for communication services. As we cannot distinguish between a failed node and an isolated node until recovery time, we treat every failure as partitioning. Partitions can occur in any number and order. Recovery of partitioning can be in a different order in which the partitioning originally occurred.

In order to provide support for partitioning, DeDiSys uses the Spread group communication and membership toolkit [ADS00]. Spread provides the extended virtual synchrony model [MAMSA94]. This model simplifies the reconciliation process of potential replication protocols, as nodes are aware which views have been installed in re-joining partitions.

We employ the *passive replication* model. In passive replication [BMST93] [GS97] requests are only processed by one *primary copy*. Updates are then propagated to the *secondary copies*. The passive model lends itself to a system where consistency is to be configured as it allows variations in the way updates are propagated. If *synchronous* update propagation is used, a primary copy must propagate any updates immediately; that is, before the result of the operation that has caused the update is returned to the client. In *asynchronous* update propagation the result is returned and the propagation of state changes performed some time later. We leave the choice of which update propagation paradigm to use to the replication protocol.

The default replication protocol [BBGME05] allows operations in all partitions to continue. Object state updates are propagated synchronously to those nodes that are reachable. If a primary copy of an object is not reachable, the protocol promotes a secondary copy to a temporary primary copy. The protocol therefore implements a "primary per partition model". The protocol also includes a reconciliation protocol that restores consistency when partitions are merged. Conflicts that occur when replicas of the same object are written to in different partitions can be resolved automatically by the replication protocol or manually by the application. However, the design of our system is such that many replication and reconciliation protocols based on the passive replication model can be implemented.

5 Replication Support Integration in CORBA

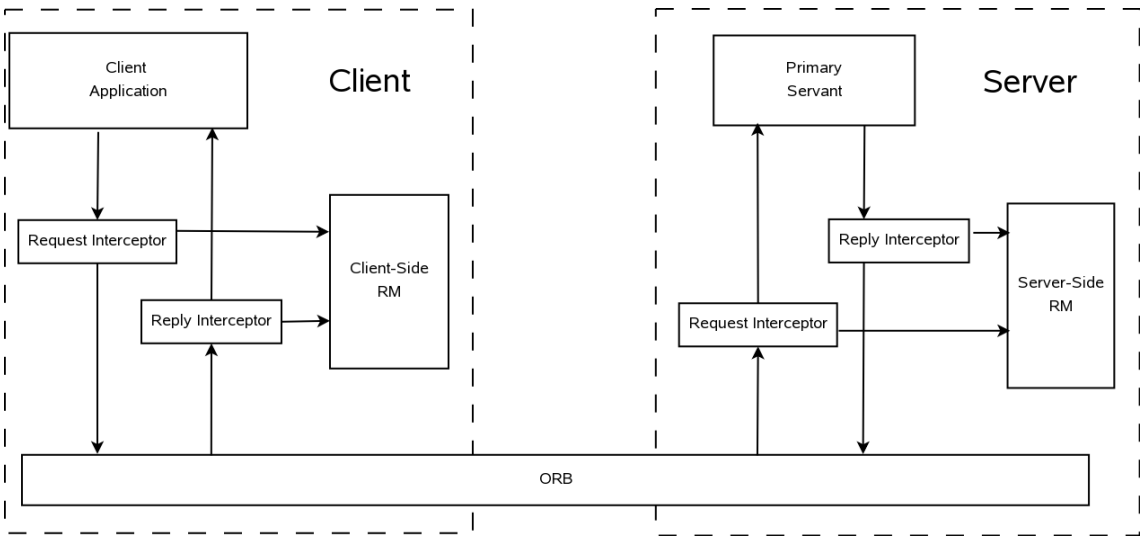

Figure 1: Replication Support Overview

Figure 1 shows how the DeDiSys replication support is integrated in CORBA. Portable interceptors are used to transfer control to the replication support, without client code having to be modified. The client invokes an object in the standard CORBA way, using a logical object reference. The DeDiSys

replication support takes care of identifying the real object reference of the primary replica. The **client-side request interceptor** is used to intercept object invocations, before they are sent. This interceptor uses the replication manager (RM) to obtain the reference of the primary replica and redirects the invocation to this primary replica. The replication manager is also used to trigger some replication protocol specific tasks that might need to be executed before the invocation can begin.

On the server side, the **server-side request interceptor** also intercepts the incoming request, in order to trigger replication protocol specific tasks. The object invocation is then executed in the standard CORBA way. Before the result is returned to the client, control is again passed to the RM. At this stage the replication protocol might require changes in the accessed object's state to be propagated to the secondary replicas of the object.

Before the request is delivered to the client application the reply is again intercepted on the client side by the **client-side reply interceptor**. At this stage a replication protocol might trigger consistency checks that could cause the invocation to be undone.

6 Object Reference Management

We distinguish between **logical object references** and **replica references**. When using the term logical object reference, we are referring to the reference of a logical object. When using the term replica reference we are referring to the actual reference of an object replica; that is, a reference of a real CORBA implementation of a logical object.

Both types of references are standard CORBA object references. However, internally logical object references only refer to an intermediate "dummy" object which is never invoked. The replication system intercepts calls to these objects and redirects them using the actual replica reference. Only logical object references are visible to client applications.

The replication support keeps track of which logical object references are associated with which replica references.

7 Replication Manager

7.1 Overview

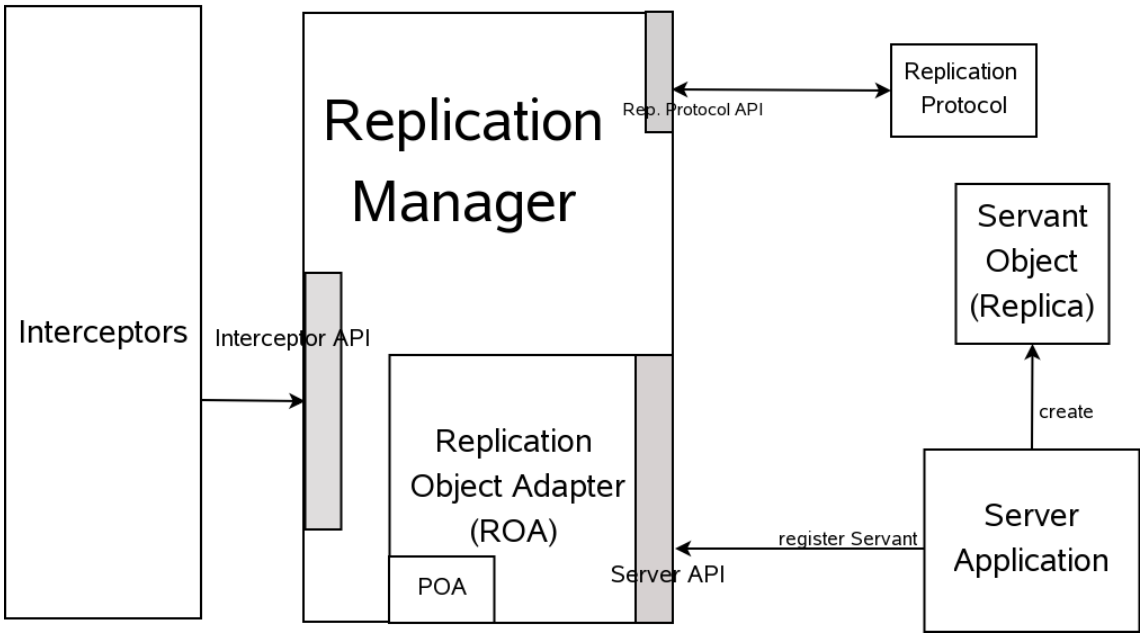

Figure 2: Replication Manager Overview

Figure 2 shows the replication manager (RM). The RM consists of various components. Only the **Replication Object Adapter (ROA)** is visible to the server application. The ROA is a CORBA object adapter. It internally uses CORBA's Portable Object Adapter (POA) and provides standard object adapter functionality, such as associating objects with object references. In addition, it manages object replicas and allows replicas to be created and associated with a logical object. Client-side-only RMs do not need a ROA.

The RM also interacts with a **replication protocol** component, in which replication protocol details, such as update transfer policies, are implemented. By encapsulating such policy in a separate component with a defined interface, the replication protocol can be changed easily.

The RM also provides an interface to the DeDiSys interceptors, in order to pass control to the replication support.

Hence the RM provides three different interfaces, which are described in detail in the next section.

Furthermore, the RM is an "application" of the Spread group membership and communication service. RMs on different nodes use Spread to exchange information on new replicas or to broadcast replica role changes. Spread is also used by the RM to keep track of which replicas are reachable. To this end, Spread callbacks handling the reception of group messages and new membership views are implemented in the RM.

7.2 Server Interface

The only interface visible to the server application is the Replication Object Adapter (ROA) interface. The ROA is a COPRBA object adapter. Internally it uses the Portable Object Adapter (POA), but adds replication functionality. The following is the interface definition of the ROA:

```
public ROA(ORB orb, ReplicationManager rm);

public org.omg.CORBA.Object createReplicatedObject (String name,

public org.omg.CORBA.Object getReplicatedObject (String name);

public org.omg.CORBA.Object registerPrimary (org.omg.CORBA.Object
                            objectToBeReplicated, Servant newReplica)
                            throws ServantNotActive, WrongPolicy;

public org.omg.CORBA.Object registerSecondary (org.omg.CORBA.Object
                             objectToBeReplicated, Servant newReplica)
                             throws ServantNotActive, WrongPolicy;

public void removeReplica (org.omg.CORBA.Object replica)

public void removeReplicatedObject (org.omg.CORBA.Object object)
```

The following example of a primary and a secondary replica being hosted on two server nodes illustrates how the ROA interface is used to create object replicas. The code for the first server is as follows:

```
org.omg.CORBA.Object objectRef = roa.createReplicatedObject
                ("obj1", "org.dedisys.exampleApp.replicatedTestObjectImpl");

replicatedTestObjectImpl r1 = new replicatedTestObjectImpl();
org.omg.CORBA.Object priRef = roa.registerPrimary(objectRef, r1);
```

First, `createReplicatedObject` is used to obtain a new logical object reference. The method takes a new string identifier for the object and the name of the object's class as arguments. The method uses the class name to create a dummy object for which a CORBA reference is obtained through the POA. This reference is returned as the logical object reference for the replicated object. The logical object reference is also mapped to the string object identifier in the RM's datastructures.

Next, a servant is instantiated and registered as a primary with the ROA using `registerPrimary`. The method associates the logical object reference with a servant. The object reference of the actual primary is returned.

On node two a secondary copy of the object can now be created:

```
org.omg.CORBA.Object objectRef = roa.getReplicatedObject("obj1");

replicatedTestObjectImpl r2 = new replicatedTestObjectImpl();
org.omg.CORBA.Object secRef = roa.registerSecondary(objectRef, r2);
```

`getReplicatedObject` is used to obtain the logical object reference for the object. Then, a new servant is created and registered with the ROA using `registerSecondary`. The method works very similar to `registerPrimary`.

Whenever the internal state of the RM has been modified, it broadcasts its new state to the the local RM's on all nodes in the system. In order to achieve synchronisation between the nodes, these RM state messages are sent as reliable atomic broadcasts.

7.3 Interceptor Interface

As described in section 5, there are four types of interceptors: the **client-side request interceptor**, the **client-side reply interceptor**, the **server-side request interceptor** and the **server-side reply-interceptor**. The RM provides an entry point for each of these interceptors to pass control to the replication support:

```
public org.omg.CORBA.Object clientPreInvocation (org.omg.CORBA.Object object);

public void clientPostInvocation (org.omg.CORBA.Object replica);

public void ServerPreInvocation (org.omg.CORBA.Object replica,
                                 ObjectClass oc,
                                 ObjectMethod om,
                                 Any[] arguments);

public void ServerPostInvocation (org.omg.CORBA.Object replica,
                                  ObjectClass oc,
                                  ObjectMethod om,
                                  Any[] arguments);
```

The `clientPreInvocation` method takes a logical object reference, and returns the replica reference of the primary object to the interceptor, so that the invocation can be redirected. `ClientPostInvocation` is called when the client receive the result of the request. The method takes the reference of the actual object that has just been invoked.

The methods `ServerPreInvocation` and `ServerPostInvocation` are the equivalent methods on the server side, but take the object's class, the actual method having been invoked and the method's arguments as additional parameters. The reason for these additional arguments is that some replication protocols might need this information.

7.4 Replication Protocol Interface

The replication protocol component is used to implement replication protocol policy that may vary between different replication protocols. Thus, the RM and the replication protocol need to provide interfaces for each other. This section describes the RM interface provided for the replication protocol. The replication protocol interface for the RM is described in section 8. The following is the definition of the interface the RM provides:

```
public org.omg.CORBA.Object locatePrimary (org.omg.CORBA.Object objectRef);

public SpreadGroup getSecondaryNodes (org.omg.CORBA.Object primaryRef);
```

```
public org.omg.CORBA.Object makeNewPrimary (org.omg.CORBA.Object objectRef);
```

The method `locatePrimary` is used to request the replica reference of the primary replica for a given logical object reference.

The *getSecondaryNodes* returns the nodes that host secondary replicas of a given primary replica. We use the Spread toolkit for group membership and communication. Therefore, the set of nodes is returned as an array of Spread group objects. The *getSecondaryNodes* method is usually used to obtain the nodes to which objects updates have to be propagated to maintain replica consistency.

Finally, the method *makeNewPrimary* provides a means to trigger the election of a new primary in case of node failure or partitioning making a primary object inaccessible. The method is typically called when `locatePrimary` has failed. It returns the replica reference of the new primary.

8 The Replication Protocol Component

8.1 Overview

The replication protocol (RP) component encapsulates replication and reconciliation policies. We locate reconciliation policy in a *ReconciliationProtocol* subcomponent of the RP, as the policies have to match each other. For instance, a replication protocol that allows updates in each of the partitions of a partitioned system requires a reconciliation policy that allows the system to recover from the inconsistencies this might introduce.

The RM passes control to the RP before and after every object invocation, in order to allow the RP to allow or deny certain object invocations to maintain consistency and to keep track of changes to objects and maintain internal data structures that hold information necessary for reconciliation. The activities of the RP in a healthy system vary from that in a system in which one or more nodes are not reachable, as different data-structures have to be maintained in these different system modes. Furthermore, the RP implements update propagation and reconciliation.

8.2 Modular Design

Different RPs can be implemented by modifying the RP component. To this end, the RP component consists of an abstract `ReplicationProtocol` class, which should be extended, in order to implement a replication protocol. `ReplicationProtocol` also provides default implementations of some methods, that may or may not be overwritten by a particular replication protocol. A default update propagation method is provided. The method can be called by any subclass implementing a specific replication protocol to broadcast the state of a specific primary replica to all secondary copies. Furthermore, a default method handling incoming replica updates is provided. This method just sets the state of all the secondary copies it holds of a particular primary copy to that included in the message. Both the update propagator and the incoming message handler can be overwritten by protocols that require more specialised implementations.

8.3 Replication Protocol Interface

The RP component provides an interface to the RM. The following is a Java interface description of the method a RP has to implement:

```
public org.omg.CORBA.Object clientPreInvoke (org.omg.CORBA.Object replica);

public void clientPostInvoke (org.omg.CORBA.Object replica);

public void serverPreInvoke (org.omg.CORBA.Object replica,
                             ObjectClass oc,
                             ObjectMethod om,
                             Any[] arguments) ;
```

```
public void serverPostInvoke (org.omg.CORBA.Object replica,
                              ObjectClass oc,
                              ObjectMethod om,
                              Any[] arguments);

public void setSystemMode (int mode);
```

The `preInvoke` methods are called by the RM before and after an object replica is invoked. These methods have the same semantics as the equivalent methods of the RM's interceptor interface described in section 7.3.

As the RM keeps track of which replicas are available it is responsible for monitoring the system mode. The system can be in normal mode or degraded mode. The RP also has to be aware of the system mode. To this end the RM can make use of the `setDegradedMode` method to inform the RP of any changes in system mode. Furthermore, the mode can be set to "reconciliation mode", which should trigger the reconciliation subcomponent to start reconciliation.

The `ReplicationProtocol` superclass implements a method providing default update propagation that can be called by any protocol implemented in a subclass:

```
void propagateUpdates (org.omg.CORBA.Object replica);
```

The method multicasts the latest state of the primary replica specified as an argument to the RPs on nodes holding secondary copies of that replica. The destination nodes of the replica update message are discovered using the RM interface described above.

8.4 The Reconciliation Subcomponent

The Reconciliation subcomponent consists of an abstract `ReconciliationProtocol` class, that has to be extended by any particular reconciliation protocol implementation. The interface provided by this abstract class is the same as that of the `ReplicationProtocol` class:

```
public org.omg.CORBA.Object clientPreInvoke (org.omg.CORBA.Object replica);

public void clientPostInvoke (org.omg.CORBA.Object replica);

public void serverPreInvoke (org.omg.CORBA.Object replica,
                             ObjectClass oc,
                             ObjectMethod om,
                             Any[] arguments) ;

public void serverPostInvoke (org.omg.CORBA.Object replica,
                              ObjectClass oc,
                              ObjectMethod om,
                              Any[] arguments);

public void setSystemMode (int mode);
```

Each of the `ReplicationProtocol` methods calls the equivalent `ReconciliationProtocol` before returning. This provides an entry points for a reconciliation protocol before and after each invocation on both the client side and the server side. Reconciliation protocols can use these entry points to update reconciliation specific data-structures during degraded mode.

9 Replication Manager Implementation

9.1 Overview

The DeDiSys replication manager has been implemented in Java. Apart from the interfaces that are provided, decision regarding the state kept in the RM and the way this state is synchronised on all nodes have had to be taken. Furthermore, certain algorithms, such as the election of a new primary copy, have had to be designed. This section describes these implementation details of our system.

9.2 Replication Manager State

The replication manager maps logical object references to replica references. To this end two tables are used. The first table maps logical object references to replica references of primary copies. The second table maps logical object references to lists of replica references of secondary copies. To provide reverse lookup a third table, mapping replica references to their logical object id, is provided.

In addition, the RM, as a client of the group membership service, keeps track of the reachable nodes in the system. These reachable nodes are maintained in a set data-structure.

Furthermore, the replication manager maps replica references to the names of the nodes that host them, in order to allow replication protocols to propagate updates to the right nodes.

Finally, every logical object reference is associated with a string name.

9.3 Algorithms

9.3.1 Joining of a New Node

When a new node joins the system, the local RM component on the joining node, needs to obtain the information contained in the existing RMs. To this end one of the existing RMs is chosen to propagate their own state to the new node. The RM uses the same mechanism for its own replication as it does for object replication. The RM in charge of propagating its state is essentially the "primary copy" of the RM.

In order to avoid complex voting protocols the primary RM is chosen according to a pre-defined order[1]. On receiving a notification from the membership service that a new node has joined, the primary RM serialises the state of its data-structures and sends it to the new node.

9.3.2 Creation of a new object

When a new object is created through the ROA the internal RM state on the node where the object has been created changes. Before the object can be invoked, this state has to be propagated to all other nodes. To this end, the state of the RM's data structures is serialised and broadcasted. In order, to synchronise the system and avoid concurrency issues when objects are created on different nodes during this process, the RM state messages are sent as reliable broadcasts with total order.

9.3.3 Election of a new Primary

When the primary copy of an object is not available due to node failure or network partitioning, a replication protocol can request the election of a new primary. To elect a new primary the RM chooses a secondary copy of the object and promotes it to a temporary primary copy. The choice of replica to promote is based on a pre-defined ordering, to avoid expensive voting.

After a local RM has changed it's state, due to the election of a new primary, it has to broadcast its new state to the other nodes in the system.

9.4 The Default Replication Protocol

DeDiSys includes a default replication protocol. The *Primary Per Partition Partition Protocol (P4)* allows operations in all partitions to continue in degraded mode. The protocol makes use of the RM's facility to promote a secondary replica to a primary replica.

When partions are re-merged a reconciliation protocol is executed. Two versions of the reconciliation protocol are provided. A manual protocol detects inconsistencies, such as updates to the same object in more than one partition, but leaves the application to remove these inconsistencies. Automatic reconciliation is also provided. In automatic reconciliation the state of the primary in one partition is imposed on all conflicting primaries in other partitions.

The protocol is described and evaluated in [BBGME05].

[1] In the current prototype implementation, simple alphabetical order based on the node name is used

10 Conclusion and Future Work

In this paper we have described the design of our fault-tolerance support for CORBA. In contrast to most approaches to fault-tolerance in CORBA and the Fault-Tolerance CORBA specification [Obj04c], the system can cope with network partitioning. The system forms part of the DeDiSys project [OFG+06], which aims at providing fault-tolerance add-ons for a variety of middlewares.

We have implemented our design in our own non-CORBA evaluation environment. The DeDiSys Lite platform [BSMEG06] serves as both a first prototype implementation of DeDiSys and an evaluation platform for replication protocols. However, it does not make use of CORBA.

We are currently implementing the architecture described here in CORBA using Java as an implementation language, taking into account the experiences gained with our non-CORBA prototype. After evaluating our implementation, the results obtained will be compared with implementations of the DeDiSys approach in other middleware architectures which are currently being developed in parallel by our project partners.

References

[ADS00] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[BBC+04] Taha Bennani, Laurent Blain, Ludovic Courtes, Jean-Charles Fabre, Marc-Olivier Killijian, Eric Marsden, and François Taïani. Implementing simple replication protocols using corba portable interceptors and java serialization. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004)*, pages 549–554, 2004.

[BBGME05] Stefan Beyer, M.C. Bañuls, P. Galdámez, and Francesc D. Muñoz-Escoí. Increasing availability in a replicated partionable distributed object system. Technical Report ITI-ITE-05/10, Instituto Tecnológico de Informática, 2005.

[BM03] Roberto Baldoni and Carlo Marchetti. Three-tier replication for ft-corba infrastructures. *Softw. Pract. Exper.*, 33(8):767–797, 2003.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. *The primary-backup approach*, pages 199–216. ACM Press, Addison-Wesley, 1993.

[BSMEG06] Stefan Beyer, Alexander Sánchez, Francesc D. Muñoz-Escoí, and Pablo Galdámez. Dedisys lite: An environment for evaluating replication protocols in partitionable distributed object systems. In *Proc. 1st International Conference on Availability, Reliability and Security*, 2006.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[FGG96] P. Felber, B. Garbinato, and R. Guerraoui. The design of a corba group communication service. In *SRDS '96: Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, page 150, Washington, DC, USA, 1996. IEEE Computer Society.

[FH02] Roy Friedman and Erez Hadad. Fts: A high-performance corba fault-tolerance service. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 61–68, Washington, DC, USA, 2002. IEEE Computer Society.

[GHN03] Fabíola Greve, Michel Hurfin, and Jean-Pierre Le Narzul. Open eden: a portable fault tolerant corba architecture. In *Proc. of the Second International Symposium on Parallel and Distributed Computing*, pages 88–95, 2003.

[GS97] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[ION94] IONA and Isis. An Introduction to Orbix+Isis, IONA Technologies Ltd. and Isis Distributed Sytems Inc., 1994.

[KOS+99] Fred Kuhns, Carlos O'Ryan, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons. The design and performance of a pluggable protocols framework for corba middleware. In *PfHSN '99: Proceedings of the IFIP TC6 WG6.1 & WG6.4 / IEEE ComSoc TC on on Gigabit Networking Sixth International Workshop on Protocols for High Speed Networks VI*, pages 81–98, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[LM97] Sean Landis and Silvano Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[LNYY03] Dongman Lee, Dukyun Nam, Hee Yong Youn, and Chansu Yu. Oci-based group communication support in corba. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1126–1139, november 2003.

[MAMSA94] Louise E. Moser, Yair Amir, P. Michael Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[MMSA+96] Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[MMSN98] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the eternal system. *Theor. Pract. Object Syst.*, 4(2):81–92, 1998.

[MSEL99] G. Morgan, S. K. Shrivastava, P.D. Ezhilchelvan, and M.C. Little. Design and implementation of a corba fault-tolerant object group service. In *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, June 1999.

[NMM97] P. Narasimhan, L. E. Moser, and P. M. Melliar.Smith. Replica consistency of corba objects in partitionable distributed systems. *Distributed System Engeneering*, 4:139–150, 1997.

[Obj04a] Object Management Group. The common object request broker architecture (corba) v.3.0.3, March 2004.

[Obj04b] Object Management Group. The common object request broker architecture (corba) v.3.0.3. chapter 11. portable interceptors, March 2004.

[Obj04c] Object Management Group. The common object request broker architecture (corba) v.3.0.3. chapter 23. fault tolerant corba, March 2004.

[OFG+06] Johannes Osrael, Lorenz Froihofer, Karl M. Goeschka, Stefan Beyer, Francesc D. Muñoz-Escoí, and Pablo Galdámez. A system architecture for enhanced availability of tightly coupled distributed systems. In *Proc. 1st International Conference on Availability, Reliability and Security*, 2006.

[RBC+03] Yansong (Jennifer) Ren, David E. Bakken, Tod Courtney, Michel Cukier, David A. Karr, Paul Rubel, Chetan Sabnis, William H. Sanders, Richard E. Schantz, and Mouna Seri. Aqua: An adaptive architecture that provides dependable distributed objects. *IEEE Trans. Comput.*, 52(1):31–50, 2003.

[RSB93] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the non partition assumption. In *IEEE Proc Fourth Workshop on Future Trends of Distributed Systems*, 1993.

[Sch93] Fred B. Schneider. What good are models and what models are good? In *Distributed Systems*, chapter 2, pages 17–26. ACM Press, Addison-Wesley, 2nd edition, 1993.

[VB98] Alexey Vaysburd and Ken Birman. The maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theor. Pract. Object Syst.*, 4(2):71–80, 1998.

[vRBH+98] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*, 28(9):963–979, 1998.

[ZMMS04] Wenbing Zhao, Louise E. Moser, and P. M. Melliar-Smith. Design and implementation of a pluggable fault-tolerant corba infrastructure. *Cluster Computing*, 7(4):317–330, 2004.