Juggler: A Management Service for CORBA Object Groups*

Marcos A. M. de Moura† and Markus Endler
Instituto de Matemática e Estatística
Universidade de São Paulo
Rua do Matão 1010
05508-900 São Paulo - SP, Brasil
Email: {mmoura, endler}@ime.usp.br

Abstract

Since its initial specification, CORBA has become a *de facto* standard for the development of object-oriented distributed applications. In spite of some important features such as support for interoperability and heterogeneity, the CORBA specification, until very recently, neither addressed object replication nor reliable communication between objects, which are key components to provide fault tolerance for distributed object systems.

Some research projects have proposed and explored a few approaches that incorporate fault tolerance mechanisms in CORBA but most of the systems developed offer a rather basic support for managing groups of object réplicas. Our work builds upon one of these systems, called OGS [3]. On the top of it, we are developing Juggler, a distributed service that provides means for flexible and automatic management of CORBA replicated objects.

1 Introduction

As the result of research in new technologies that facilitate the development of distributed systems, some architectures that support object-oriented distributed programming in open environments have emerged. One of such architectures is CORBA [12], specified by OMG to provide an object-oriented infrastructure that makes possible communication between objects. CORBA emphasizes some aspects such as code reuse, systems integration and heterogeneity, but until very recently it did not address some fundamental problems of distributed systems (eg partial and temporary failures in the system and consistent event ordering). However, in March 2000, OMG has finished the technology adoption vote for Fault Tolerant CORBA (FT-CORBA) [13], which standardizes CORBA functionalities supporting fault tolerant applications.

*Supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) - Proc. No. 98/06138-2.

†Graduate student supported in part by CNPq.

7 Fault Tolerant CORBA

Recently, OMG has adopted a specification for Fault Tolerant CORBA (FT-CORBA) [13], which aims to provide support for applications that require high degree of reliability and availability. This specification basically defines interfaces for replication management, fault detection, notification and fault recovery.

The *ReplicationManager* interface allows an application to add and remove members of an object group, as well as to get location of the group members, ie object réplicas. This interface also provides means to set the replication, membership and consistency styles of a group, and its initial and minimum number of réplicas. The counterpart of the *ReplicationManager* in Juggler's architecture is the *GroupManager* interface. However, in our project we are implementing protocols that support both active and primary-backup replication, as well as weak consistency replication and means to dynamically switch between them, which is not specified in FT-CORBA.

The fault management in FT-CORBA is provided by a very general and well designed architecture, which encompasses fault detectors, notifiers and fault analyzers. In Juggler we make use of the *Monitoring Service* of OGS in order to detect faults. These faults are propagated to the *GroupObserver* objects, which act as the fault notifiers in Juggler's architecture. In our project we do not make any provision to support fault analyzers. Finally, a major difference between the architecture proposed by Juggler and that of FT-CORBA is that we do not specify mechanisms for stable storage logging, thus supporting only a very simple recovery mechanism. At last, we should note that the main goal when developing Juggler is to create a flexible management service for CORBA object groups and not a generic fault tolerance infrastructure such as the one recently adopted by OMG.

8 Conclusion

Support for fault tolerance is an usual requirement of many real-world dependable distributed applications. Recently, some systems have been designed to support the development of fault tolerant, highly available applications in CORBA, but they only provide a minimal set of mechanisms for dealing with object groups.

Through our work we aim to provide high-level abstractions that ease the management of fault tolerant CORBA applications. Juggler supports monitoring and failure recovery of object réplicas, availability management and specification of several management policies for fault tolerant applications. The main contributíons of Juggler are the replication of the information concerning object groups and a flexible specification of replication policies for CORBA replicated objects. An interesting feature of our service is that the Juggler-OGS layers are wholly portable and interoperable with different CORBA ORBs.

Despite the high cost of managing OGS groups when compared to other systems that offer similar features, eg Electra and Eternal, we have chosen it as the main replication management system in the software architecture adopted in project SIDAM. This decision was taken mainly due to the assumption that our computational environment may integrate different platforms and CORBA ORBs and also because the applications that will use the SIDAM architecture do not have high-performance requirements.

We have implemented and tested an initial prototype of Juggler through which we have evaluated its performance and identified the major overheads associated with the

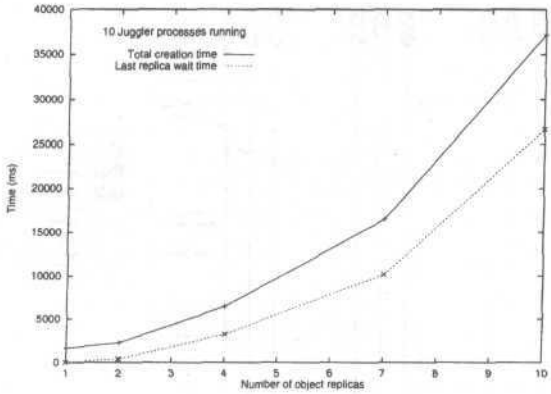

Figure 9: The time spent when creating object groups through Juggler

The other test (see Figure 10) aimed at identifying the cost of the individual steps performed by Juggler during the group creation protocol, and the influence that the number of Juggler processes running had on this cost. The graph in Figure 10 shows the results for creating of a group with 2 members. Notice that the values shown in this graph are accumulated. For instance, the "Supports multicast" values encompass the time spent by the *supports* multicast itself and the time spent to create the object réplicas (represented by the line "Réplicas creation time"). From the graph it can be seen that the total creation time grows at a small rate as the number of Juggler processes grows. Although our test was only for two members and up to 10 Juggler processes, we expect this rate to be similar when the number of Juggler processes becomes larger. We can also see that the cost of the several multicasts issued among Juggler's objects accounts for a small fraction of the overall group creation time, and that this cost does not grow significantly as the number of Juggler processes increases.

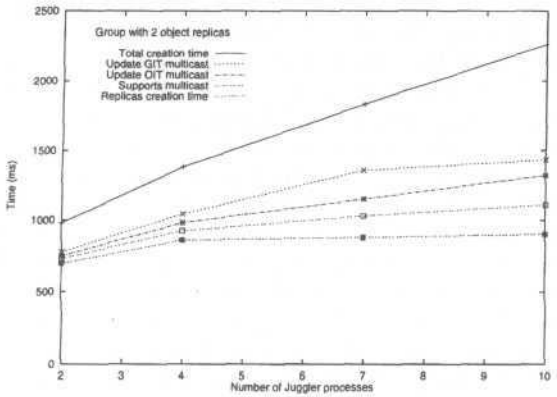

Figure 10: The cost of the steps performed during group creation

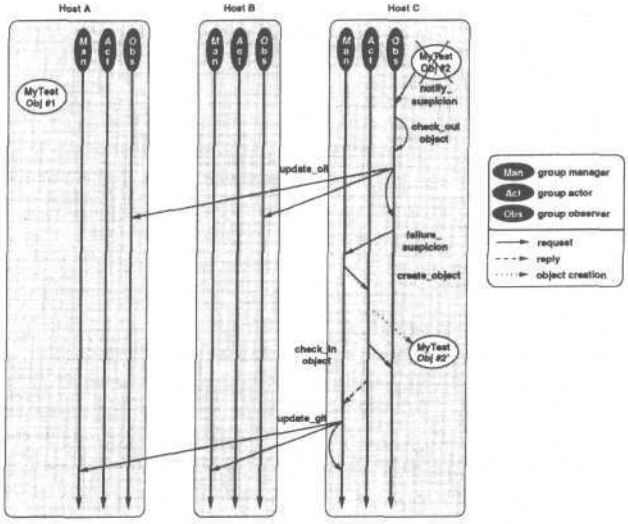

Figure 8: Failure recovery protocol

has been extensively tested with this CORBA ORB. We chose Java mainly because of code portability, thus allowing the use of Juggler in different platforms without code modification.

An initial prototype of Juggler has been developed in order to test the group creation protocol. For this purpose we implemented a CORBA object that represents a bank account and then we replicated it using Juggler so as to achieve high availability. These objects define methods to deposit and withdraw money from a bank account, and a method to get the current balance, which is the replicated state between the object réplicas. The tests were performed on a local 10 Mbit Ethernet network interconnecting 12 Sun SPARCstations 4, running Solaris 7, shared also by other users. The tests were run with the TCP_NODELAY option set, which forces the immediate sending of requests, instead of buffering them and sending them in batches.

In the first test (see Figure 9) we measured the total time spent by Juggler to create object groups with one to ten members (object réplicas). For this test we have instantiated 10 Juggler processes, each running on a different host. From the graph we can see that as the number of réplicas in the group grows, the time needed to create the group grows at a higher rate. For instance, groups with 2 and 4 réplicas were created in approximately 2.3s and 6.4s respectively, whereas in average a group with 10 réplicas took 37.2s to be created. This happens mainly because the access to the CORBA Naming Service (used by OGS during the creation of object réplicas) is done, in our current prototype, via mutual exclusion, so as to guarantee that the réplicas have a consistent group view upon group instantiation. The lower line in the graph represents the time that the last object replica of the group had to wait until it could register itself with CORBA Naming Service and join the group. Hence, it can be noticed that the sequential access to this service is the determinant factor of the total creation time of an object group, and that the effective overhead introduced by Juggler's creation protocol is relatively small.

automatically reconfigured according to the changes in the computational environment, such as overload of some hosts or other network problems.

Changes of the group configuration that affect only the number or the location of the object réplicas are implemented by having Juggler automatically create and/or destroy object réplicas on the hosts specified in the *Hosts* attribute of the group semantics. In order to support also changes of the replication style we have extended the *Group Administrator* class made available by OGS. Essentially, our *Group Administrator* implements the protocols for different replication styles (active, primary-backup, weak consistency) and a means to dynamically switch between them without causing heavy impact on the access time to the corresponding group (ie only a short period of time in which requests to the group are blocked). In fact, this extension does not require any other interfaces that application objects must implement, except the essential interface for receiving multicast messages and notifications of view changes.

Thus, if the system administrator decides to change the replication style of a given object group, Juggler will make a copy of the semantics associated with the group, set the *ReplicationStyle* attribute accordingly and invoke the **change_group** operation on any *GroupManager* object made available by Juggler. The *GroupManager* then multicasts a message to the extended *Group Administrator* objects, which will then block new requests from any *Group Accessor* and switch to the specific protocol required to manage message delivery and view changes according to the new replication style.

Failure Recovery of Object Réplicas

In this section we will describe the actions performed during the automatic recovery of a failed object replica. Suppose that the object replica, of the "MyTest" group, located at *Host C* fails. Remember that the semantics associated with the "MyTest" group, which is shown in Figure 6, specifies automatic failure recovery and a minimum of 2 object réplicas. Figure 8 illustrates the failure recovery protocol, discussed as follows.

As soon as the object replica fails the OGS Monitoring Service invokes the **notify_suspicion** operation on the local *GroupObserver* (ie located at *Host C),* which causes a local call to the **check_out_object** operation. This operation removes the failed object from the *Object Information Table,* through an **update_oit** multicast on the *OBSERVERS* group. The *GroupObserver* forwards the failure suspicion to the local *GroupManager,* which checks the semantics associated with the group of the failed object. The *GroupManager* then instructs the local *GroupActor* to create a new replica to substitute the failed object. The local *GroupActor* creates the object, invokes the **check_in_object** operation on the local *GroupObserver,* which updates the *Object Information Table* (through an **update_oit** multicast on the *OBSERVERS* group, not shown in Figure 8), and returns a reference to the new object replica to the local *GroupManager.* The *GroupManager* then updates the information concerning the group of the failed object in the *Group Information Table,* through an **update_git** multicast on the *MANAGERS* group.

6 Prototype Implementation

In order to develop Juggler we have chosen Visibroker for Java 3.2 and the Java language. Visibroker was selected due to its vast set of features and also because OGS

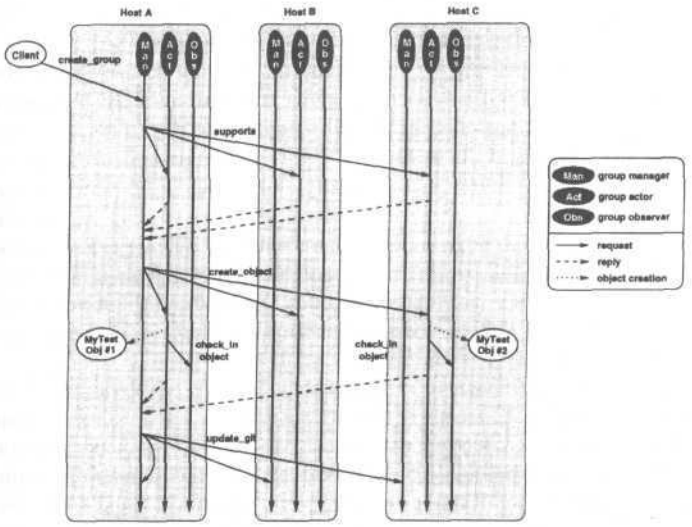

Figure 7: Group creation protocol

Initially, a client application invokes the **create_group** operation on any *GroupManager* object made available by Juggler (eg through the CORBA Naming Service). The *GroupManager* unpacks the information contained in the *Semantics* object and checks whether there are enough hosts so that the *InitialReplicationDegree* attribute can be satisfied and if the *Hosts* attribute contains only valid system hosts, ie hosts in which Juggler is currently running.

Next, the *GroupManager* multicasts a **supports** message to the *ACTORS* group to know if the object réplicas can be created on the specified hosts. Then, the **create_object** operation is invoked on the *ACTORS* group, passing the *TypeId*, *Hosts* and *Criteria* as arguments. Each *GroupActor* checks whether the host in which it is running is specified in the *Hosts* argument. If this is not the case, no further action is taken. Otherwise, a new object replica is locally created, the **check_in_object** operation is invoked on the local *GroupObserver* (which updates the *Object Information Table* through an **update-oit** multicast on the *OBSERVERS* group, not shown in Figure 7) and a reference to the new object replica is returned to the *GroupManager* that requested the **create_object** operation. Finally, the *GroupManager* receives the references to the object réplicas created and stores all the information concerning the new group in the *Group Information Table*, through an **update_git** multicast on the *MANAGERS* group.

Modification of Group Configuration

Juggler provides means for on-the-fly modification of replication policies (semantics) for object groups. This is an important feature of our service, since it can deal with applications whose requirements change over the time, such as an object's replication degree or the form of synchronization among object réplicas. Moreover, this feature supports adaptive fault tolerance in the sense that applications managed by Juggler may be

InitialReplicationDegree specifies the initial number of réplicas to be created in an object group and *MinimumReplicationDegree* determines the minimum number of réplicas that are required for the given group. *ReplicationStyle* indicates the form of replication used in the object group, eg primary-backup, active replication and weak consistency[1]. The administrator of a fault tolerant application can configure Juggler to automatically recover from failures of object réplicas by setting *AutomaticRecovery* to *true.* Another option is to set this attribute in such a way that Juggler only produces notifications whenever an object replica fails, letting the system administrator decide on the most suitable actions to be taken. *TypeId* attribute specifies the repository identifier for the type of the object réplicas. Finally, the attribute *Hosts* indicates where the object réplicas of the group must be created; if it is left blank Juggler will chose randomly where to create the object réplicas. In Figure 6 we show how to specify a group *Semantics.*

```
// Java
Juggler.Semantics sem = new Semantics();
// Defines the Sémantics attributes
sem.initialReplicationDegree = 2;
sem.minimumReplicationDegree = 2;
sem.replicationStyle = Juggler.ReplStyle.ACTIVE;
sem.automaticRecovery = true;
sem.typeId = "IDL:BankAccount/Account:1.0";
sem.hosts = new String[2];
sem.hosts[0] = "A";
sem.hosts[1] = "C";
```

Figure 6: Specification of a group *Sémantics*

5.2 Management Protocols

In this section we will discuss the protocols used in Juggler for group creation, failure recovery of object réplicas and modification of the configuration of an object group. In order to implement these features, Juggler defines three OGS groups, namely *MANAGERS, ACTORS* and *OBSERVERS.* These groups comprise, respectively, all the *GroupManager, GroupActor* and *GroupObserver* objects running in the system network. The *MANAGERS* group maintains a replicated *Group Information Table,* which contains information about all the object groups, such as their names, semantics and members. The *ACTORS* group is responsible for the creation and destruction of object réplicas. The *OBSERVERS* group maintains two replicated tables, namely *Host Information Table* and *Object Information Table,* which contain information about all the hosts where Juggler is running and all the object réplicas created at these hosts, respectively.

Group Creation

We will now describe how Juggler components interact in order to create a new object group called "MyTest", whose semantics is the one specified in Figure 6. In order to simplify this example, we will assume that no errors will occur during the creation of the group. The necessary actions performed during this task are illustrated in Figure 7.

[1] The weak consistency replication style is not completely defined yet.

on an object factory. The object factory implements the *GenericFactory* interface, which is like the *GroupActor* interface, except that the *hosts* parameter does not appear in both **create_object** and **destroy_object** operations. Object factories must be implemented by the application programmer, whose responsibility is therefore to code the operations concerning the objects' life cycle. This frees the *GroupActor* object from this obligation, raaking it a more general object factory. We assume that each *GroupActor* object retains a reference to an object factory that is able to manage the objects' life cycle on the host where the *GroupActor* is located.

4.2.3 *GroupObserver* interface

This is an internai interface of Juggler. It provides operations for keeping information about currently active hosts and object réplicas. This interface inherits from the *Notifiable* interface specified by OGS, which only defines the **notify_suspicion** operation. The IDL specification of the *GroupObserver* interface is shown in Figure 5.

```
// IDL
interface GroupObserver : OGS::Notifiable {
    void arriving_host(in string name);
    void departing_host(in string name);
    void check_in_object(in Host host, in TypeId typeId, in Object obj);
    void check_out_object(in Object obj);
    ObjectList    active_objects();
};
```

Figure 5: IDL specification of the *GroupObserver* interface

The **arriving_host** operation is invoked to signal that the host identified by the *name* parameter is ready to manage object groups. Similarly, the **departing_host** operation is invoked to signal that the host identified by the *name* parameter cannot manage object groups any longer. The **check jn_object** and **check_out_object** operations enable the *GroupObserver* to have a precise information about which objects are running on which hosts of the system. The **active_objects** operation returns information about all existing object réplicas, such as their types, locations and references. Finally, the **notify_suspicion** operation, which is the unique operation inherited from the *OGS::Notifiable* interface, allows the *GroupObserver* to receive notification about failure suspicion of object réplicas.

5 Group Management in Juggler

In this section we focus on the main constructs and protocols used to implement Juggler's management of object groups.

5.1 Group Configuration and Semantics

In order to support flexible management of object groups in Juggler we have designed a structure called *Semantics*, which contains several criteria that guide the creation and the configuration of object groups. The main attributes defined in this structure are as follows.

5.1) specified by the *sem* parameter and with application-specific criteria, such as initialization values for the object réplicas, specified by the *crit* parameter. If a group with the given name already exists the *GroupExists* exception is raised. *InvalidSemantics* and *CannotMeetSemantics* exceptions are raised if the semantics specified for the group is invalid or if it cannot be enforced. The **change_group** and **destroy_group** operations are used to modify the semantics and the criteria associated with an object group, and to destroy an object group respectively. If there is no group with the given *name* the *NoSuchGroup* exception is raised.

When the **destroy_host** operation is invoked all the object réplicas running in the host identified by *name* are destroyed. If the *keep_running* parameter is set to *false* then the Juggler process running in the specified host is also destroyed, otherwise Juggler is kept running. *NoSuchHost* exception is raised if the specified host does not exist, ie there is no Juggler process running on it. *CannotDestroyHost* exception is raised if some object replica currently active in the specified host cannot be destroyed, thus preventing Juggler from violating the semantics associated with some object group. The **active_groups** operation returns information about existing object groups, such as their names, members and semantics. Finally, the **active_hosts** operation returns the name of the hosts in which Juggler is currently running.

4.2.2 *GroupActor* interface

This is an internai interface of Juggler, meaning that it is not externally visible to application objects. It provides operations for creation and destruction of object réplicas. The IDL specification of the *GroupActor* interface is shown in Figure 4.

```
// IDL
interface GroupActor {
    boolean supports(in TypeId typeId);
    Object create_object(in Typeld typeld, in HostList hosts, in Criteria crit)
        raises(NoFactory, ObjectExists);
    void destroy_object(in Typeld typeld, in HostList hosts)
        raises(NoSuchObject);
};
```

Figure 4: IDL specification of the *GroupActor* interface

The **supports** operation checks whether it is possible to create an object whose repository identifier (a *CORBA::RepositoryId* type) for the type of the object is specified in the *typeId* parameter. When the **create_object** operation is invoked a new object identified by the *typeId* parameter is created, but only if the *GroupActor* is located in a host specified in the *hosts* parameter. In addition, application-specific criteria may be specified through the *crit* parameter. *NoFactory* and *ObjectExists* exceptions are raised if there is no factory to create the object and if an instance of the requested object is already running on the host, respectively. If this operation succeeds then a reference to the created object is returned to the callee. Finally, the **destroy_object** operation destroys the object specified by the *typeId* parameter, but only if the *GroupActor* is located in a host specified in the *hosts* parameter. If the requested object does not exist the *NoSuchObject* exception is raised.

It is important to note that the *GroupActor* object, which implements the *GroupActor* interface, does not create or destroy objects itself, but invokes the appropriate operations

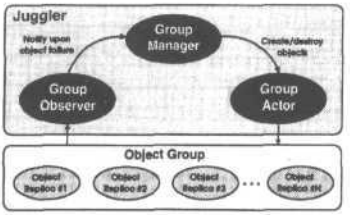

Figure 2: Juggler logical architecture

GroupManager mainly deals with group creation, destruction and on-the-fly modification of the group configuration. Furthermore, this object invokes methods on *GroupActor*, which is responsible for creation and destruction of object réplicas. *GroupObserver* is essentially responsible for failure suspicions of object réplicas and the delivery of these notifications to the *GroupManager*, which takes actions to maintain the desired configuration for the related group according to a semantics provided by the application programmer.

4.2 Interfaces

Juggler is specified as a collection of CORBA IDL interfaces, namely *GroupManager*, *GroupActor* and *GroupObserver*. These interfaces, implemented by objects named *GroupManager*, *GroupActor* and *GroupObserver*, respectively, are described as follows.

4.2.1 *GroupManager* interface

This interface provides the externai view of Juggler for the application's programmer. It basically provides operations for group management and for information retrieval concerning currently active object groups and hosts in the system. The IDL specification of the *GroupManager* interface is shown in Figure 3.

```
// IDL
interface GroupManager {
    void create_group(in string name, in Semantics sem, in Criteria crit)
        raises(GroupExists, InvalidSemantics, CannotMeetSemantics);
    void change_group(in string name, in Semantics sem, in Criteria crit)
        raises(NoSuchGroup, InvalidSemantics, CannotMeetSemantics);
    void destroy_group(in string name)
        raises(NoSuchGroup);
    void destroy_host(in string name, in boolean keep_running)
        raises(NoSuchHost, CannotDestroyHost);
    GroupList active_groups();
    HostList active_hosts();
};
```

Figure 3: IDL specification of the *GroupManager* interface

The **create_group** operation is invoked by a Juggler client to create a new group of object réplicas, uniquely identified by *name*, with a replication semantics (see Section

GroupObserver, and their relations with other software layers, is shown in Figure 1.

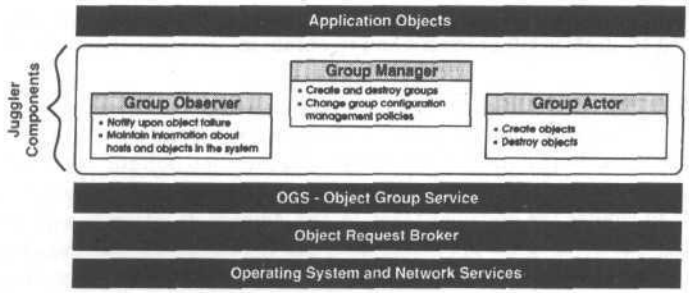

Figure 1: Juggler architecture

Juggler only uses externally visible components of OGS, namely the *Monitoring Service* and the *Object Group Service.* Juggler uses the *Monitoring Service* in order to receive notifications about failure suspicions of monitored objects (ie group members). The *Object Group Service* is responsible for all operations related to group management (ie group creation/destruction and addition/removal of members), consistency maintenance and reliable group communication.

In OGS, the management of object groups on the server side is implemented through *Group Administrator* objects, which are associated with each of the application object's replica in the group. In addition, these objects interact with each other to execute the consensus and voting protocols. From the perspective of the application object, each *Group Administrator* object is responsible only for delivering multicast messages and notifying group view changes to the group member attached to it. On the client side, OGS provides *Group Accessor* objects (ie proxies), which enable clients to issue invocations to the object replicas' methods, specify the number of replies to be expected and the desired reliability of each method invocation, without even knowing how many réplicas exists, and where they are executing. Thus, it is the *Group Accessor* which does the multicast to the corresponding *Group Administrator* objects, and eventually waits for a reply from them.

The main contributions of our work are the definition of a high-level interface that eases the management of CORBA applications based on replicated objects and an implementation which replicates the information about object groups. This replication is a key feature of Juggler since it makes possible to recover from object, host and service failures, thus providing a robust, fault tolerant management service for CORBA objects. Juggler's architecture is similar to that of Piranha, mainly because of its basic management features and the distribution of information concerning object groups. However, Juggler provides means for specifying flexible management policies for groups of object réplicas, which is best tackled in Proteus.

4.1 Overview

Logically we can think of each object group as being managed by a Juggler process, composed of three objects, namely *GroupManager*, *GroupObserver* and *GroupActor*, and which is itself replicated. The interaction between these objects is shown in Figure 2.

voting schemes as well as mechanisms for fault detection and notification. *QuO runtime* is used to specify *Quality of Service (QoS)* requirements for an application and an *Object Factory* is used to start or kill processes and to obtain information about the hosts of the system.

We can distinguish two main components in the architecture of Proteus, namely an *Advisor* and a *Protocol Coordinator*. The *Advisor* receives *QoS* requests transmitted via the *QuO runtime* and fault notifications regarding application objects. It then determines the most appropriate configuration for the related application. The *Protocol Coordinator* is responsible to enforce the decisions taken by the *Advisor*. It interacts with *Object Factories* in order to start or to kill processes. Through the use of the *Gateways* the *Protocol Coordinator* sets the appropriate replication style, type of voting, degree of replication and type of faults to be supported by an application, according to the decision taken by the *Advisor*.

Piranha and Proteus have both very interesting set of features but due to some architectural or implementation decisions they do not fully provide suppórt for all the management requirements mentioned in the beginning of this section. Piranha is based on a very robust architecture, since information about all the objects running in the system is replicated, enabling the service to recover all the objects that were executing in a failed host and to continue its normal functioning. A good feature of this service is its suppórt for automatic failure recovery, but it still lacks adequate mechanisms for adaptation and dynamic modification of application-specific availability policies, eg replication style and fault types to be supported. Proteus provides a good suppórt for automatic failure recovery and mechanisms that allows it to dynamically choose how to provide the fault tolerance required by an application. Unfortunately, Proteus neither implements functionalities that enable dynamic choice of faults to tolerate nor protocols to dynamically switch between replication styles. Moreover, Proteus dependability manager is not replicated, which makes it a very fragile management service.

4 Juggler

Juggler is a management service for groups of CORBA object réplicas. It is intended to suppórt automatic reconfiguration and recovery of object groups and to provide means for a flexible specification of management policies for fault tolerant CORBA applications. These policies include several properties, such as number of réplicas, replication style (active replica, primary-backup and weak consistency), automatic failure recovery, and others.

Juggler is based on OGS [3], a service that supports object groups in CORBA. OGS provides basic suppórt for creating and destroying object groups, adding and removing group members and protocols for view change and for state transfer. We have chosen OGS as the basis of our service mainly because it has mechanisms for specifying different communication and replication styles for each object group and also due to its suppórt for monitoring and failure notification of individual objects within object groups. Furthermore, OGS provides group communication using only CORBA standardized constructs, which *guarantees portability* and *interoperability* in this architecture. An architectural sketch of Juggler illustrating its main components, ie GroupManager, GroupActor and

3 Related Work

The incorporation of group communication mechanisms in CORBA is a useful and efficient approach to provide reliability and fault tolerance for this architecture. However, most systems that support these features only offer basic operations to manage objects groups, and do not provide a high-level interface that can be used to handle more complex replica management policies.

Distributed applications management typically requires support for automatic failure recovery, adaptation to changes in the system environment and dynamic modification of the application requirements by the system administrator. In order to provide this kind of support for fault tolerant CORBA applications, some extra mechanisms and services based on group communication were embedded in or added to some systems. More specifically, we will describe how Piranha [8] and Proteus [14] provide some of these aforementioned features in the Electra system and in the AQuA architecture, respectively.

3.1 Piranha

Piranha is a service composed of a collection of objects that implement notification, failure detection and restart services for CORBA applications. Piranha acts as a network monitor, notifying failures in replicated objects. It also works as an application manager, providing support for automatic restart of failed objects, on-the-fly replication of stateful objects, object migration and maintenance of a certain replication degree for CORBA applications.

This service was built to run over Electra, a CORBA ORB that provides support for object groups, reliable multicast, state transfer, failure detection and virtual synchrony. Piranha user's interface shows the applications running on each host, their configuration parameters, error messages and other kinds of notifications. Several activation criteria can be specified during creation of replicated objects, such as hosts where to create the réplicas, number of réplicas to be created and their life time, and the architecture, load and performance of the hosts where the réplicas must be created. More advanced criteria inform whether objects should be swapped out after a certain period of time without processing requests and if Piranha should automatically restart failed objects.

3.2 Proteus

Proteus provides fault tolerance in the AQuA architecture by supporting dynamic management of replicated distributed objects. One of its functions is to decide how to provide fault tolerance for an application based on the availability requirements specified by the application's programmer. This decision involves choosing the replication style (active or passive), type of voting (algorithm and location), degree of replication and type of faults to tolerate (crash, value and time) in order to configure the application according to the desired availability. The other function of this dependability manager is to support dynamic modification of the configuration of an application, based on the decision about the most adequate fault tolerance scheme to support.

Proteus interacts very closely with other AQuA components, namely *Gateway*, *QuO runtime* and *Object Factory*. A *Gateway* maps IIOP messages into Ensemble messages and vice versa. It provides an infrastructure for implementing different replication and

Moreover, all the operations used to manage object groups are embedded in the ORB so that any object in the system can become a group member.

In systems that adopt this approach, eg Electra [7] and Orbix+Isis [5], CORBA requests assigned to object groups are passed to a group communication system, which is responsible for transmitting them via reliable, ordered multicast to the group members. The main advantages of this approach are replication transparency to client applications and good performance of group communication. However, by modifying the ORB, it makes these systems difficult to interoperate with other CORBA ORBs. In addition, fault tolerant applications developed on top of these systems are inherently not portable among different ORBs.

2.2 Interception Approach

Using the interception approach [9] all the mechanisms that provide group communication become hidden from the ORB. The main idea here is to intercept the IIOP requests generated by the ORB and map them onto a reliable, ordered multicast provided by a groupcommunicationsystem.

The interception of the messages can be done through the use of meta-object protocols, operating systems interfaces or mechanisms provided by CORBA. Two systems that implement the interception approach are the Eternal system [10] and the AQuA architecture [1]. This approach has the advantage of providing replication in a transparent way to application objects, which allows that already existing applications benefit from fault tolerance support. Another advantage is interoperability, since a system that adopts this approach can choose to intercept only those requests sent to object groups, letting the ORB deal with requests to singleton objects. A disadvantage of this approach is loss of portability, since specific mechanisms of the operating systems can be eventually explored. In addition, similarly to the integration approach, there is a high dependence on the group communication toolkits being used.

2.3 Service Approach

In the service approach [4] group communication is provided as a CORBA service [11] that runs above the ORB, instead of being part of it. In this case, the ORB is not aware of object groups but the use of the group service becomes explicit to client applications. This may not be considered a real drawback of the approach, since a system can generate smart proxies in order to hide the use of the group service from client applications. Two systems that adopt this approach are OGS [3] and OFS [15].

Unlike the interception approach, where legacy applications can benefit from fault tolerance support in a transparent way, with the service approach any group member must implement specific callback operations in order to receive notifications about group view changes and to process state transfer requests. The main advantages of this approach are full portability, interoperability and compliance with the CORBA standard, since neither proprietary group communication toolkits nor the use of specific operating systems interfaces are required.

Since the early 80's many research projects have been focusing on solving problems related to partial failures and on aiming to guarantee predictable behavior of distributed programs. Most models proposed by these works are usually based on replication of processes or objects and on providing reliable group communication. With group communication, distributed program's behavior becomes predictable even in case of failures, when asynchronous communication is used and when processes or objects join or leave the system dynamically. Thus, group communication can be considered a powerful paradigm for the development of fault tolerant and highly available distributed applications.

Recently, many efforts have been made to make possible the incorporation of group communication mechanisms in CORBA. In general, the systems resulting from these works provide a CORBA environment enhanced with reliable and predictable communication, as well as mechanisms to deal with partial failures and consistent event ordering. However, these systems only provide a rather low-level interface for the application programmer, in the sense that he or she has to deal with all the burden of replication, eg failure recovery and management of object groups. In order to facilitate these tasks we are developing Juggler, a distributed service that provides means for flexible and automatic management of fault tolerant CORBA applications, instead of providing a generic fault tolerance infrastructure such as the one recently adopted by OMG.

Juggler is a key element of the software infrastructure being developed in project SIDAM [2] (Distributed Information Systems for Mobile Agents). This project aims at studying the software architecture and development issues involved in the implementation of distributed information services to be accessed by mobile clients. The application model proposed in SIDAM was motivated by a real-life application: an on-line service providing traffic information for a metropolitan area such as São Paulo. In this context, Juggler will be used to manage replicated instances of data objects containing traffic information.

The remainder of this paper is organized as follows. Section 2 presents an overview of the main approaches to incorporate group communication mechanisms in CORBA. Section 3 describes related works. Section 4 gives an overview of Juggler's architecture and its component interfaces. A description of the main structures and protocols used in Juggler is given in Section 5. In Section 6 some implementation issues are discussed. In Section 7 we make a brief comparison between Juggler and FT-CORBA. Finally, Section 8 summarizes and concludes the paper.

2 Group Communication in CORBA

The incorporation of group communication in CORBA is a natural means of providing fault tolerance to CORBA applications. The systems that offer such support can be classified according to the way that group communication mechanisms are incorporated in this environment. Thus, we can distinguish three main approaches to achieve this goal, namely integration, interception and service, discussed as follows.

2.1 Integration Approach

The basic idea of the integration approach [6] is to modify the ORB so that, internally, it is able to distinguish references to singleton objects from references to object groups.

group creation protocol. We are now improving the access to the CORBA Naming Service, implementing features that allow the interpretation and dynamic modification of the group semantics and developing protocols for primary-backup and weak consistency rcplication styles.

It is important to note that the cost of the group management service offered by Juggler will be payed exclusively by the applications that use it, thus neither compromising nor degrading the overall performance of the other applications. Moreover, Juggler introduces only little overhead to fault tolerant applications since it only has effect during the initial creation of object groups and when the replication style for a group is changed.

References

[1] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pagcs 245 253, West Lafayette, Indiana, USA, October 1998. IEEE.

[2] D. M. da Silva, M. D. Gubitoso, and M. Endler. Sistemas de Informação Distribuídos para Agentes Móveis. In *Proceedings of the XXIV Brazilian Software and Hardware Seminars (SEMISH '98)*, Belo Horizonte, Brazil, August 1998. SBC. Available in http://www.ime.usp.br/~dilma/papers/semish98.ps.

[3] P. Felber. *The CORBA Object Group Service: A Service Approach to Objecl Groups in CORBA.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

[4] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the 15th Symposium on Re.liable Distributed Systems (SRDS '96)*, pages 150 161, Los Alamitos, CA, USA, October 1996. IEEE Computer Society Press.

[5] Isis Distributed Systems Inc. and IONA Technologies Ltd. Orbix+Isis Programmer's Guide, 1995. Document D070-00.

[6] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice. of Object Systems*, 1997.

[7] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, University of Zurich, Switzerland, 1995.

[8] S. Maffeis. Piranha: A CORBA Tool for High Availability. *IEEE Computer*, 30(4):59–66, April 1997.

[9] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, Portland, OR, USA, June 1997.

[10] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica Consistency of CORBA Objects in Partitionable Distributed Systems. *Distributed Systems Engineering Journal*, 4(3):139-150, September 1997.

[11] OMG. CORBAservices: Common Object Services Specification. Technical Report , Object Management Group, December 1998.

[12] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report Formal/99-10-07, Object Management Group, 1999. Version 2.3.1.

[13] OMG. Fault Tolerant CORBA: Joint Revised Submission. OMG, January 2000. Document orbos/2000-01-19.

[14] B. S. Sabnis. Proteus: A Software Infrastructure Providing Dependability for CORBA Applications. Master's thesis, University of Illinois, USA, 1998.

[15] G. Sheu, Y. Chang, D. Liang, S. Yuan, and W. Lo. A Fault-Tolerant Object Service on CORBA. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, 1998.