An Infrastructure for Management of Fault Tolerant CORBA Applications*

Marcos A. M. de Moura†
Instituto de Matemática e Estatística
Universidade de São Paulo
Rua do Matão, 1010
05508-900 São Paulo - Brazil
mmoura@ime.usp.br

Markus Endler
Departamento de Informática
PUC-Rio
Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro - Brazil
endler@inf.puc-rio.br

PUC-RioInf.MCC42/01, Dezembro 2001

Abstract

In the last years 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, only very recently CORBA specification, with the adoption of the Fault Tolerant CORBA standard (FT-CORBA) [10], addressed issues concerning object replication and reliable communication between objects, which are key components to provide fault tolerance for distributed object systems.

Since mid-90´s several research projects have proposed and explored some approaches to incorporate fault tolerance mechanisms in CORBA but most of the systems developed offer a rather basic support for managing groups of object replicas. Our work builds upon one of these systems, called OGS [4]. We created an extended OGS (called OGS+) and on the top of it we developed Juggler, a distributed service that provides means for flexible and automatic management of CORBA replicated objects through the deployment of a set of high-level interfaces based on the ones standardized by FT-CORBA.

Keywords: Fault Tolerance, CORBA, Object Groups, Replication, Replica Management

Resumo

Nos últimos anos CORBA tornou-se um padrão *de fato* para o desenvolvimento de aplicações distribuídas orientadas a objeto. Apesar de tratar de alguns aspectos importantes tais como apoio à interoperabilidade e heterogeneidade, só recentemente a especificação CORBA, através do padrão (FT-CORBA) [10], enfocou assuntos como a replicação de objetos e a comunicação confiável entre objetos, que são componentes chave para prover tolerância à falhas em sistemas de objetos distribuídos.

Desde meados dos anos 90 diversos projetos de pesquisa propuseram e exploraram algumas abordagens para incorporar mecanismos para tolerânca à falhas em CORBA, mas a maioria dos sistemas desenvolvidos oferece somente um suporte básico para gerenciar grupos de objetos replicados. Nosso trabalho se baseia em um destes sistemas, chamado OGS [4]. Em nosso trabalho implementamos uma extensão de OGS (chamanda OGS+) e em cima deste sistema desenvolvemos Juggler, que é um serviço distribuído para o gerenciamento automático de grupos de réplicas de objetos CORBA. Este gerenciamento se dá através de interfaces de alto nível baseadas no padrão FT-CORBA.

Palavras-chave: Tolerância à Falhas. CORBA, Grupos de Objetos, Gerenciamento de Réplicas

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

†The author was supported in part by CNPq and is currently working at Claudino S.A. - Lojas de Departamentos, Teresina - PI, Brazil. E-mail: mmoura@armazemparaiba.com.br

1 Introduction

Throughout the last two decades computers have been widely used in critical applications, such as air traffic control, nuclear energy stations' management, patients vital signal monitoring, etc. In these environments, a single failure or even just a temporary unavailability of a computer system can have a disastrous consequence to the life of human beings as well as to the their surrounding environment. Thus, there is a demand for systems that guarantee different kinds and degrees of Quality of Service (QoS) to the applications, such as reliability and availability, which are key aspects of fault tolerance.

Along with the large acceptance and use of the *Common Object Request Broker Architecture* (CORBA) [7], there has been also an increasing demand for a standard specifying functionalities that enable the development of fault tolerant CORBA applications. In April 2000 OMG adopted the first version of such standard, called Fault Tolerant CORBA or just FT-CORBA [10].

The specification of FT-CORBA defines a set of interfaces, policies and services that facilitate the development of highly reliable CORBA applications which avoid a single point of failure. The standard is based on the use of replicated objects (object groups) and describes a set of fault tolerance strategies and mechanisms for the detection, notification and analysis of faults on object replicas. Furthermore, FT-CORBA also specifies mechanisms that support automatic failure recovery and control of the consistency of data shared among object replicas within an object group.

In this paper we describe a FT-CORBA-like fault tolerance infrastructure, called Juggler/OGS+ [3], that allows for the development and management of CORBA applications requiring high degree of reliability and availability. Juggler/OGS+ is composed of two main services, namely Juggler and OGS+. OGS+ is an extension of OGS [4], a programming and execution environment for fault tolerant CORBA applications that allows the creation and use of object groups. Juggler can be seen as the management tool for applications developed with OGS+. It provides a set of high-level group management interfaces and implements mechanisms similar to the ones proposed by OMG in the FT-CORBA standard.

Juggler allows the use of serveral fault tolerance strategies that define the behaviour of object groups. It allows the use of different replication styles and fault tolerance properties, such as the location and number of object replicas in a group, on a per-group basis. Moreover, it provides mechanisms that makes possible to dynamically modify the fault tolerance properties as well as the replication style of object groups. Thus, Juggler provides the system administrator with a fine grained control over the execution of fault tolerant CORBA applications, so that applications can be better adapted to the computational environment in which they are executed.

The remainder of this paper is organized as follows. In the next section we present an overview of the FT-CORBA standard. Section 3 describes some related works. Section 4 gives an overview of Juggler/OGS+'s architecture and its component interfaces. In Section 5 some implementation issues, tests and results are discussed. Finally, Section 6 summarizes and concludes the paper.

2 Fault Tolerant CORBA

Recently, OMG has adopted a specification for Fault Tolerant CORBA, called FT-CORBA [10], which aims to provide support for applications that require a high degree of reliability and availability. This specification basically defines interfaces for replication management, fault detection, notification and fault recovery.

FT-CORBA standard offers great flexibility for object groups' management. It supports several fault tolerance strategies, such as active and passive replication, request redirection and transparent method reinvocation when passive replication is used. FT-CORBA defines mechanisms that allow for the detection, notification and analysis of faults on object replicas. Moreover, the standard supports automatic failure recovery, checkpointing and logging mechanisms, which allow the fault tolerance infrastructure to perform automatic control of the consistency between the object replicas that compose a group.

FT-CORBA standard defines foure main components: the *ReplicationManager*, the *FaultNotifier*, the *GenericFactory* and the *FaultDetector*, which must be implemented as CORBA objects. FT-CORBA also determines that failure recovery, checkpoint and logging mechanisms be not CORBA objects, but that they are to be implemented as part of the ORB or as a service that works in between the ORB and the operating system.

2.1 Replication Management

The *ReplicationManager* component allows an application to add and remove members of an object group, as well as to get the locations of the group members, i.e. object replicas. This component also provides means to set parameters such as the replication style, membership and consistency styles of a group, as well as its initial and minimum number of replicas. In order to provide fault tolerance *ReplicationManager* can be replicated, but logically there is only one such component in each Fault Tolerance Domain.

ReplicationManager inherits the following interfaces: *PropertyManager*, *GenericFactory* and *ObjectGroupManager*. Interface *PropertyManager* allows to choose the fault tolerance properties associated to an object group. Some of these properties can be modified dinamically, such as the initial and the minimum number of replicas in a group. Other properties, such as the replication, consistency and membership styles used by a group cannot be modified after group creation.

FT-CORBA rules that a group´s membership and consistency be controlled by the fault tolerance infrastructure or by the application itself. Moreover, the standard defines the following replication styles: STATELESS, COLD_PASSIVE, WARM_PASSIVE and ACTIVE. STATELESS replication should be used for groups in which the behaviour of the group is not affected by the history of invocations. When ACTIVE replication is used all the group members receive and execute every request sent to the object group. In this replication style duplicated requests and replies are suppressed by the fault tolerance infrastructure.

In COLD_PASSIVE and WARM_PASSIVE replication styles only one member of the group (primary) executes the methods invoked on the group, while the other members are backups of the primary replica. The main difference between these replication styles is that when a failure of the primary occurs, the full state of the primary is retrieved from a log and loaded into a backup, if COLD_PASSIVE is used. If WARM_PASSIVE is used then a backup, whose state is periodically updated by the primary, is promptly elected the new primary.

Interface *GenericFactory* is used by an application to create and destroy object groups. This interface is also used by the *ReplicationManager* to create and destroy the object replicas that compose a group. The other interface inherited by the *ReplicationManager*, called *ObjectGroupManager*, allows to add or to remove an object to an object group. This interface also enables an application to define the primary replica of a group and to obtain other information related to the group, such as the location and IOR of each member of the group.

2.2 Fault Management

Fault management in FT-CORBA is structured in a very general architecture which encompasses three components: *FaultDetector*, *FaultNotifier* and *FaultAnalyzer*. The *FaultDetector* is responsible for detecting faulty object replicas using the *pull* monitoring model [10].

Component *FaultNotifier* provides operations to register objects interested in receiving notifications concerning failure events. This interface also defines operations that enable the transmission of these notifications to the consumers of fault events. The *FaultAnalyzer* component establishes the correlation between fault events and condensed fault reports. Fault analysis is clearly application-specific and therefore the FT-CORBA standard does not define an interface for the *FaultAnalyzer*.

2.3 Failure Recovery and Logging

During normal processing of an object group using passive replication, the logging mechanism stores the state and the actions processed by the primary member of a group in a log. If the group uses WARM_PASSIVE replication style this mechanism is responsible for periodically sending the updates performed on the primary member´s state to the backup members. If the group uses COLD_PASSIVE replication then the logging mechanism is used to store the full state of the primary replica of the group.

In case of failure of the primary replica of a group the fault tolerance infrastructure promotes one of the backup members to become the new primary replica. In this case, the failure recovery mechanism updates the state of the new primary replica according to the information stored on the log. Thus, the failure recovery mechanism guarantees that a previous backup member promoted to be the new primary replica has a consistent state stored before its starts processing new requests sent to the group. The failure recovery mechanism is also used during the addition of new replicas to a group that adopts active replication, so that all group members always share the same state.

2.4 Object Groups Addressing Scheme

FT-CORBA defines a special IOR, called *Interoperable Object Group Reference* (IOGR), which is used to address an object group. An IOGR contains multiple TAG_INTERNET_IOP profiles, which are used to access the object replicas that compose a group. Each TAG_INTERNET_IOP profile must contain a TAG_FT_GROUP profile, which stores information that allows to access an object group. It stores an identifier for the object group and the fault tolerance domain in which the group is defined. Each TAG_INTERNET_IOP can also contain a TAG_FT_PRIMARY component, which can be used, for example, to access the primary replica in an object group that adopts a passive replication style.

2.5 Fault Tolerance Domains

In order to provide means for managing arbtrarily large and complex applications, FT-CORBA defines Fault Tolerance Domains. A domain tipically contains several hosts and object groups. A host can be part of more than one fault tolerance domain and all groups of objects within a domain are created and managed by a unique logical *ReplicationManager*. Furthermore, FT-CORBA allows that an object group to invoke and to be invoked by objects defined in other domains.

2.6 Limitations of the Standard

Despite standardizing a rich set of interfaces, services and mechanisms to manage fault tolerant CORBA applications, FT-CORBA standard has some limitations concerning the use of client applications that use already existent CORBA ORBs and concerning interoperability between the fault tolerance infrastructures developed by different vendors.

FT-CORBA allows that a client developed in an ORB that does not implement this standard invoke operations on replicated objects. However, such client cannot take full benefit from the fault tolerance properties provided through the use of object groups. Moreover, FT-CORBA requires that within each Fault Tolerance Domain all the ORBs and fault tolerance infrastructures used must be from the same vendor in order to guarantee interoperability and the use of all the fault tolerance properties within the domain.

3 Related Works

3.1 Group Communication in CORBA

Since mid-90´s the incorporation of Group Communication in CORBA has become a natural means for supporting object replication in CORBA. The systems that offer such support can be classified according to the way that Group Communication mechanisms are incorporated. Three main approaches can be distinguished: integration, interception and service.

The basic idea of the integration approach [12] is to modify the ORB so that, internally, it is able to distinguish references to singleton objects from references to object groups. In this approach, 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. Some systems that implement this approach are Electra [14] and Orbix+Isis [11].

Using the interception approach [16] all the mechanisms that provide Group Communication are 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 Group Communication system, thus providing total replication transparency to CORBA applications. Two systems that implement the interception approach are the Eternal system [17] and the AQuA architecture [2]. The aforementioned approaches always require the use of a proprietary Group Communication toolkit.

In the service approach [5] Group Communication is provided as a CORBA service [6] that must be accessed through the ORB, but that is not a 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 [4] and OFS [21]. 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 portability, interoperability and compliance with the CORBA standard, since neither proprietary Group Communication toolkits nor the use of specific operating system interfaces are required.

3.2 Fault Tolerance Management

Along with the emergence of the first systems that allow the development of fault tolerant CORBA applications through the use of Group Communication, some high level tools to manage CORBA object groups arised, such as Piranha, Proteus and, more recently, DOORS.

Piranha [15] is a service, built on the top of Electra CORBA ORB [14], that is 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 of replicated objects. It also works as an application manager, providing support for automatic restart of failed objects, dynamic replication of stateful objects, object migration and enforcing a given replication degree for CORBA object groups. 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, thus, to continue its normal functioning. However, this service still lacks adequate mechanisms for adaptation and dynamic modification of application-specific availability policies, e.g. the replication style and the fault types to be supported.

Proteus [19] provides fault tolerance in the AQuA architecture [2] by supporting dynamic management of replicated distributed objects. One of its functions is to decide how to provide fault tolerance for applications based on *availability requirements* specified by the application's programmer, which consists of 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. So far, 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.

DOORS [18] is an infrastructure that supports fault tolerance in CORBA which implements a large subset of the functionalities, services and mechanisms standardized by FT-CORBA. In fact, the development of DOORS began before the adoption of FT-CORBA and many concepts and components defined in this service influenced the standard adopted by OMG. Thus, DOOR's architecture is very similar to that defined in FT-CORBA. However, this service does not implement ACTIVE replication style, does not provide mechanisms to allow automatic failure recovery and logging and does not provide a fault notification service, as proposed in FT-CORBA.

4 Juggler/OGS+

Juggler/OGS+ aims at providing a complete infrastructure for the development and management of fault tolerant CORBA applications based on the use of object groups. Juggler/OGS+ is a combination of two services, namely Juggler and OGS+. Juggler is mainly concerned with offering a high-level interface to manage replicated objects (object groups), while OGS+ is responsible for maintaining the structure and consistency of object groups.

Figure 1 shows Juggler/OGS+'s architecture. From the figure we can see that application objects can access Juggler and OGS+ directly. Those that access Juggler can use a more intuitive, flexible and high-level service for managing object groups which makes tasks such as replica monitoring and failure recovery transparent to the applications. On the other hand, those accessing OGS+ directly are exposed to low-level mechanisms for management of object groups.

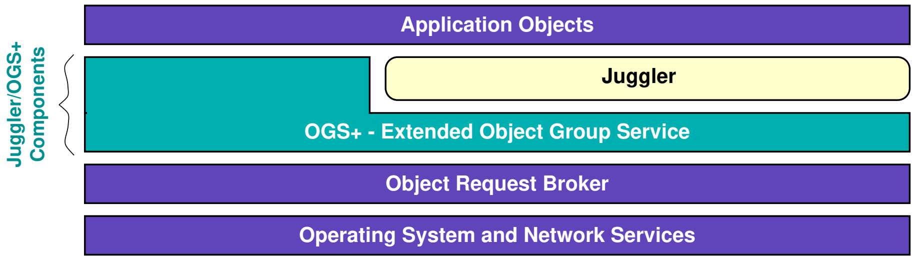

Figure 1: Juggler/OGS+ architecture

Although Juggler and OGS+ are clearly separated at architectural level and have their own set of fuctionalities and interfaces these services work very closely at implementation level, as discussed in the following sections.

4.1 Juggler

Juggler is a service that provides a robust infrastructure for management of CORBA applications that require a high degree of reliability and are based on replicated objects. This service provides interfaces for configuration management of fault tolerant CORBA applications which are based on object redundancy (object groups), and also mechanisms for the detection and recovery of failed objects.

Through Juggler, it is possible to define the replication style and the fault tolerance properties associated to each object group individually. These properties define, for instance, the location and the minimum number of replicas that compose a group, and allow also to choose the most suitable form of failure recovery within a group. Juggler supports the creation and management of object groups with different replication styles, such as active replication [20], primary-backup replication [1] and a variant of virtual-primary-copy [13]. It also allows the dynamic modification of the fault tolerance properties and replication style associated to each object group.

Juggler is built upon OGS+ (see Section 4.2), an extension of OGS [4], a service that supports object groups in CORBA. OGS provides basic support for creating and destroying object groups, adding and removing group members, as well as protocols for view change and for state transfer. We have chosen OGS as the basis of our infrastructure mainly because it has mechanisms for specifying different communication and replication styles for each object group and also due to its support 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.

Juggler makes extensive use of two main components of OGS+, namely the *Monitoring Service* and the *Object Group Service*. It uses the *Monitoring Service* to receive notifications about failure suspicions of monitored objects (i.e. group members) and the *Object Group Service* to manage object group´s life cycle and to provide reliable group communication. Juggler service is composed of three main components: `GroupManger`, `GroupActor` and `GroupObserver`, which are specified as collections of OMG IDL interfaces and may be implemented as groups of replicated objects (see Section 4.1.3). As shown in Figure 2, Juggler can be seen as a unique logical entity that is composed of these three components working close together.

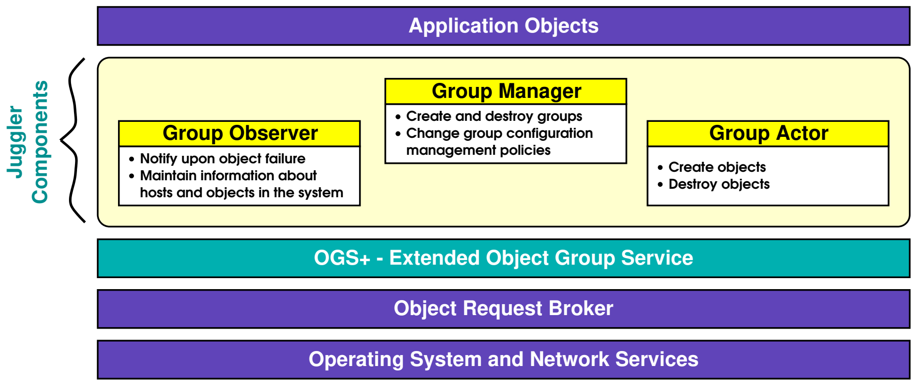

Figure 2: Juggler architecture

`GroupManager` mainly deals with group creation, destruction and dynamic modification of the group configuration. Furthermore, this component invokes methods on `GroupActor`, which is responsible for creation and destruction of object replicas. `GroupObserver` is essentially responsible for failure suspicions of object replicas and delivery of these notifications to the `GroupManager`, which takes actions to maintain the requested availability for the related group according to a semantics provided by the system administrator.

4.1.1 Group Semantics

Each object group created through Juggler has its fault tolerance properties stored in a structure called `Semantics`, which is shown in Figure 3. Attribute `type_id` contains the identification of the object replicas within CORBA Interface Repository [7]. Attribute `the_locations` stores a list of the locations where the group members are executing. A location is a physical host, or cluster, where object replicas can be created. For each group, only one replica can exist at a given location.

Other attributes are `initial_number_replicas` and `minimum_number_replicas`, which hold, respectively, the initial and minimum number of replicas that should exist in the group. `Minimum_number_replicas` only makes sense if `automatic_recovery` attribute is set, which indicates

```
struct Semantics {
    FT::TypeId type_id;
    LocationSeq the_locations;
    FT::InitialNumberReplicasValue initial_number_replicas;
    FT::MinimumNumberReplicasValue minimum_number_replicas;
    AutomaticRecoveryValue automatic_recovery;
    ReplicationStyle replication_style;
};
```

Figure 3: `Semantics` structure

that Juggler must automatically recover failed object replicas. Otherwise an external entity must control the minimum number of object replicas within the group. Finally, attribute `replication_style` stores the replication style associated to the group and the specific properties related to the it. Juggler supports the following replication styles: STATELESS, COLD_PASSIVE, WARM_PASSIVE, ACTIVE and VIRTUAL_PRIMARY_COPY, which are discussed in Section 4.2.1.

4.1.2 Juggler Components

GroupManager

The `GroupManager` component basically provides means for the creation and destruction of object groups, and operations to modify and to obtain their associated semantics. It implements the main interface of Juggler, called `GroupManagementOperations`, and other interfaces that make possible for it to be replicated and to receive notifications concerning failed object replicas. The `GroupManagementOperations` interface, shown in Figure 4, provides functionalities similar to the ones defined by FT-CORBA in `ObjectGroupManager` interface.

```
interface GroupManagementOperations {
    GroupInfo create_group(in ObjectGroup object_group,
                          in Semantics the_semantics,
                          in FT::Criteria the_criteria)
          raises(GroupAlreadyExists, InvalidSemantics, CannotMeetSemantics);
    GroupInfo add_members(in ObjectGroup object_group,
                          in LocationSeq the_locations,
                          in FT::Criteria the_criteria)
          raises(FT::ObjectGroupNotFound, FT::MemberAlreadyPresent, CannotMeetSemantics);
    GroupInfo remove_members(in ObjectGroup object_group,
                          in LocationSeq the_locations)
          raises(FT::ObjectGroupNotFound, FT::MemberNotFound, CannotMeetSemantics);
    GroupInfo set_replication_style(in ObjectGroup object_group,
                          in ReplicationStyle the_replication_style)
          raises(FT::ObjectGroupNotFound, mGroupAdmin::BadReplicationStyle);
    GroupInfo set_minimum_number_replicas(
                          in ObjectGroup object_group,
                          in FT::MinimumNumberReplicasValue minimum_number_replicas)
          raises(FT::ObjectGroupNotFound, CannotMeetSemantics);
    GroupInfo set_recovery_style(in ObjectGroup object_group,
                          in AutomaticRecoveryValue automatic_recovery)
          raises(FT::ObjectGroupNotFound);
    void delete_group(in ObjectGroup object_group)
          raises(FT::ObjectGroupNotFound, CannotMeetSemantics);
};
```

Figure 4: `GroupManagementOperations` interface

Operation **create_group** is used to create a new object group. Its first parameter, `object_group`, is a string that uniquely identifies the object group to be created. The semantics associated to the group is specified in parameter `the_semantics`. The last parameter, `the_criteria`, allows to pass application-specific information used during the creation of object replicas.

Add_members and **remove_members** operations are used, respectively, to add and to remove members of an object group. Juggler also specifies operations that allow to change other fault tolerance properties. **Set_replication_style** operation allows the modification of the replication style associated to a group, which can be modified while the group is working. The protocol that describes how this operation is performed is discussed in Section 4.2.2.

Operation **set_minimum_number_replicas** is used to specify the minimum number of replicas that should exist in a group and **set_recovery_style** allows to determine whether Juggler is responsible for automatic recovery of failed group members. Finally, **delete_group** operation is used to destroy an object group. `GroupManagementOperations` interface also defines other five operations (not shown here). These operations are used to obtain the location and a reference for individual group members and to retrieve information concerning the object groups managed by Juggler.

GroupActor

The `GroupActor` component is responsible for creation and destruction of object replicas. These operations are inherited from the `GenericFactory` interface, which is shown in Figure 5. This interface is exactly the same as defined in FT-CORBA standard. `GenericFactory` interface must also be implemented by the application programmer in order to allow the creation of objects of different types at several locations. This interface, as implemented by `GroupActor`, only allows the creation of object replicas in the locations where Juggler is running. In fact, the `GroupActor` component does not create or destroy objects, it only delegates these operations to the Object Factories registered with Juggler service. This frees the `GroupActor` from dealing with the creation of different types of object replicas, thus making it a generic Object Factory.

```
interface GenericFactory {
    typedef any FactoryCreationId;
    Object create_object(in TypeId type_id,
                         in Criteria the_criteria,
                         out FactoryCreationId factory_creation_id)
          raises(NoFactory, ObjectNotCreated, InvalidCriteria,
                 InvalidProperty, CannotMeetCriteria);
    void delete_object(in FactoryCreationId factory_creation_id)
          raises(ObjectNotFound);
};
```

Figure 5: `GenericFactory` interface

`GenericFactory` interface only defines two operations: **create_object** and **delete_object**. **Create_object** operation allows to create an object whose type is specified in `type_id` parameter. `The_criteria` parameter is used to specify additional criteria during the creation of an object, such as start-up parameter initialization and the location of the object. Parameter `factory_creation_id` returns an unique identifier for the created object that must be used further to delete the object through the **delete_object** operation.

In addition to the `GenericFactory` interface, `GroupActor` also implements `FactoryOperations` interface, which allows to add and remove the registry of application-specific Object Factories that will be responsible for carrying on **create_object** and **delete_object** operations.

GroupObserver

In Juggler service, the operations concerning object replica monitoring and failure suspicion of replicas are defined in two interfaces inherited by the `GroupObserver` interface, namely

`ReplicaMonitoring` and `FaultNotifier`, which are shown in Figure 6. `FaultNotifier` interface offers a subset of the functionalities provided by the CORBA Notification Service, commonly known as `CosNotification` [8], and is identical to the one specified in FT-CORBA [10] with the modifications proposed in [9].

```
interface ReplicaMonitoring {
    void start_monitoring_replica(
                in FT::GenericFactory::FactoryCreationId factory_creation_id);
    void stop_monitoring_replica(
                in FT::GenericFactory::FactoryCreationId factory_creation_id);
};

interface FaultNotifier {
    typedef unsigned long long ConsumerId;
    void push_structured_fault(in CosNotification::StructuredEvent event);
    void push_sequence_fault(in CosNotification::EventBatch events);
    ConsumerId connect_structured_fault_consumer(
                in CosNotifyComm::StructuredPushConsumer push_consumer);
    ConsumerId connect_sequence_fault_consumer(
                in CosNotifyComm::SequencePushConsumer push_consumer);
    void disconnect_consumer(in ConsumerId connection)
        raises(CosEventComm::Disconnected);
    void replace_constraint(
                in ConsumerId connection,
                in CosNotification::EventTypeSeq event_types,
                in string constr_expr);
};
```

Figure 6: `ReplicaMonitoring` and `FaultNotifier` interfaces

`ReplicaMonitoring` interface defines **start_monitoring_replica** and **stop_monitoring_replica** operations, which allow the `GroupObserver` to start and to stop monitoring object replicas created by Juggler through the `GroupActor`. `FaultNotifier` interface makes it possible for the `GroupObserver` to forward notifications concerning failed object replicas to the `GroupManager`, which is responsible to recover them. Objects that wish to receive these notifications must register with `FaultNotifier` through **connect_structured_fault_consumer** or **connect_sequence_fault_consumer** operation.

Push_structured_fault and **push_sequence_fault** operations are used to send notifications, respectively, as structured events (`CosNotification::StructuredEvent`) or event batches (`CosNotification::EventBatch`), to the event consumers registered with Juggler. Operation **disconnect_consumer** is used to disconnect an event consumer from the `FaultNotifier` and the **replace_constraint** operation allows a consumer to specify which types of events it wants to receive from the `FaultNotifier`.

In addition to these interfaces, `GroupObserver` also implements the `LocationOperations` interface, which allows to update and retrieve information concerning the locations where Juggler is running. Typical informations are the load and performance of the location, which can be set to any metrics defined, for instance, by a *Resource Manager*. These informations are used by Juggler when choosing a location to create an object replica.

ReplicationManager

Besides defining `GroupManager`, `GroupActor` and `GroupObserver` components, Juggler also defines a component, called `ReplicationManager` (see Figure 7), whose purpose is to aggregate all the operations implemented in those components that must be made available to Juggler clients. It may be used not only by usual Juggler clients, but it also serves as the basis for the implementation of

Juggler client interfaces, e.g. a Graphical User´s Interface.

```
interface ReplicationManager :  GroupManagementOperations,
                                  LocationOperations,
                                  FactoryOperations {
    void register_fault_notifier(in FT::FaultNotifier fault_notifier);
    FT::FaultNotifier get_fault_notifier()
         raises(FT::InterfaceNotFound);
};
```

Figure 7: `ReplicationManager` interface

`ReplicationManager` only defines two operations, namely **register_fault_notifier** and **get_fault_notifier**, where the first operation is used to register `FaultNotifiers` with Juggler service and the second allows to retrieve an IOR for a `FaultNotifier` previously registered with the service. After obtaining such reference, a client of Juggler can also register itself with the `FaultNotifier` in order to receive notifications of events generated by Juggler.

`ReplicationManager` component also allows a Juggler client to access the operations defined in `GroupManagementOperations`, `FactoryOperations` and `LocationOperations`, which are implemented, respectively, by `GroupManager`, `GroupActor` and `GroupObserver` and were discussed throughout this section.

4.1.3 Group Management in Juggler

In this section we will discuss the protocols used in Juggler for group creation, failure recovery of object replicas 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* (GIT), which contains information about all the object groups, such as their names, semantics and list of members. Since the *ACTORS* group is responsible for the creation and destruction of object replicas, it stores the *Factory Information Table* (FIT), which contains information of all Object Factories registered with Juggler. Another replicated structure stored in *ACTORS* group is the *Object FactoryCreationId Map Table* (OFMT), which contains a mapping between the IOR of each object replica and its corresponding FactoryCreationId. The *OBSERVERS* group also maintains two replicated tables, namely *Location Information Table* (LIT) and *Object Information Table* (OIT), which hold information about all the hosts where Juggler is running and the monitoring of object replicas created at these hosts, respectively.

The tables maintained by *MANAGERS*, *ACTORS* and *OBSERVERS* groups are updated through the use of operations defined, respectively, in *GroupManager*, *GroupActor* and *GroupObserver* interfaces. The names of these operations are very intuitive and they will be mentioned during the description of the protocols for group creation and failure recovery, discussed as follows.

Group Creation

In order to facilitate the understanding of the group creation protocol (shown in Figure 8), we describe the steps during the creation of an object group named "Test", which has two object replicas, located at hosts "A" and "C", and that adopts active replication.

Initially, a client application invokes the **create_group** operation on any *ReplicationManager* object made available by Juggler/OGS+ through the CORBA Naming Service. This object only delegates this operation to one of the *GroupManager* objects. Then, the *GroupManager* invokes **create_object** operation on the *ACTORS* group and waits for the replies from all of its members. Each member of the *ACTORS* group checks whether an object replica must be created locally. This is the case for hosts "A" and "C", where the registered Object Factories create the replicas. The *GroupActor* objects that created a replica then invoke **add_object_factoryId** operation on the *ACTORS* group, which is used to update the OFMT.

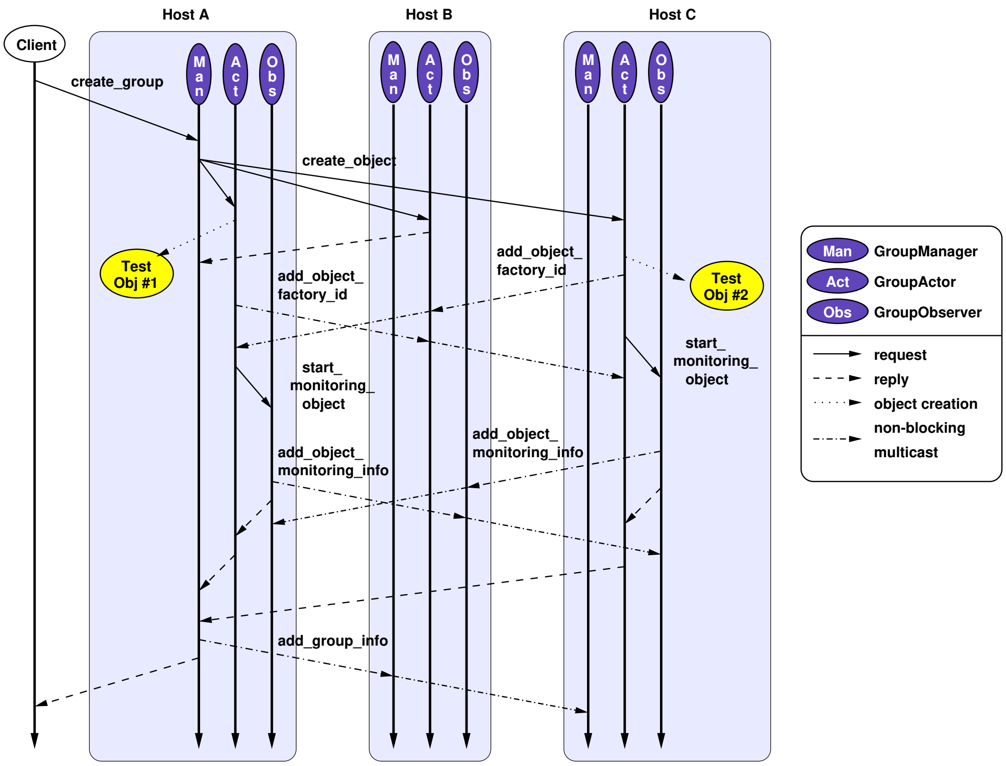

Figure 8: Group creation protocol

Next, each *GroupActor* invokes the **start_monitoring_object** operation on the *GroupObserver* instantiated on the same host in which it is being executed. Then, the *GroupObservers* asynchronously invoke **add_monitoring_info** operation on the *OBSERVERS* group, thus updating the OIT. Each *GroupActor* waits for the conclusion of the **start_monitoring_object** operation and returns a reference to the *GroupManager* that started the protocol. After receiving all the replies, the *GroupManager* stores the information concerning the group and disseminates it by asynchronously invoking the **add_group_info** operation on the *MANAGERS* group, which updates the GIT. Finally, the *GroupManager* object that started the protocol returns the information concerning the new object group to the client that requested the operation.

Failure Recovery

Juggler implements a protocol that allows for automatic failure recovery of object replicas, which is illustrated in Figure 9. To facilitate the description of this protocol we will use group "Test" again, assuming that it was created as described in the previous section. Now let us suppose that a failure occurs in the object replica located on host "C" and that the semantics of the group specifies that Juggler should automatically recover failed objects.

When an object replica fails, the *GroupObserver* receives a notification concerning the failure through the Monitoring Service offered by OGS+. Next, the *GroupObserver* invokes **remove_object_monitoring_info** operation on *OBSERVERS* group (updating the OIT), and sends a notification about the failure to the *MANAGERS* group. One of the *GroupManagers* is deterministically chosen to carry on the failure recovery procedure. Then, the chosen *GroupManager* creates

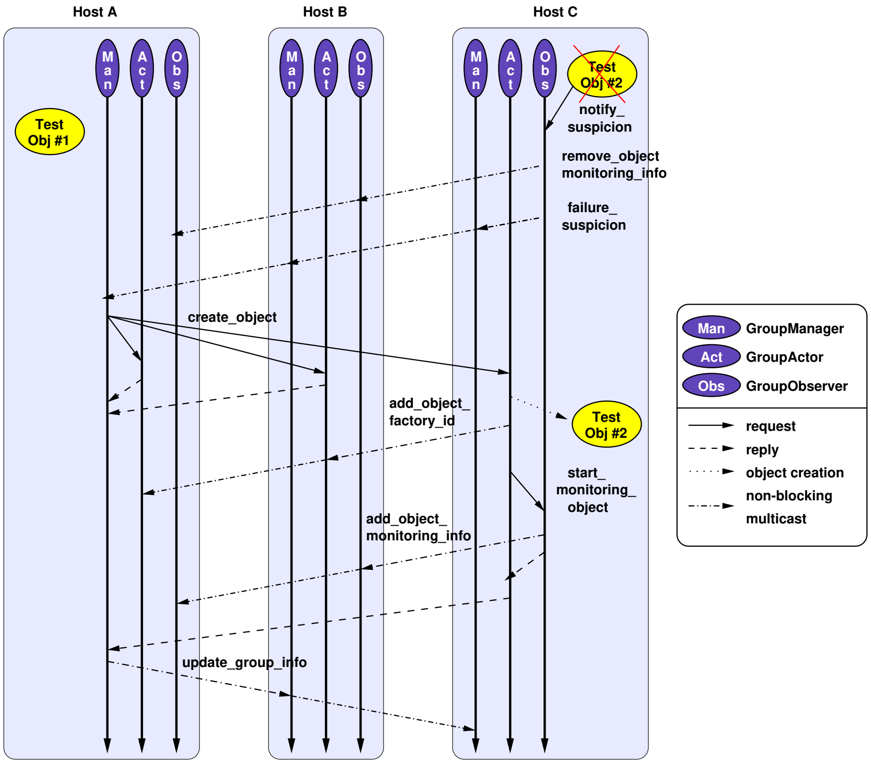

Figure 9: Failure recovery protocol

a new replica on the same host where the failed one was running by invoking the **create_object** operation on the *ACTORS* group. From this point on, the failure recovery proceeds as the creation of a single object replica, which only differs from the protocol described in the previous section in that the last operation to be invoked on the *MANAGERS* group is the **update_group_info**, which updates the *Group Information Table* (GIT).

Modification of the Group Semantics

Juggler provides means for dynamic modification of the semantics associated to object groups. This makes it possible for Juggler to deal with applications whose requirements, such as the number of object replicas or the replication style of an object group, change over the time. Moreover, this feature supports adaptive fault tolerance in the sense that applications managed by Juggler may be automatically reconfigured due to the changes in the computational environment, such as overload of some hosts or other network problems.

Changes of the group configuration (semantics) that affect only the number or the location of the object replicas are implemented by having Juggler automatically create and/or destroy object replicas on the hosts specified in the `the_locations` attribute of the group semantics. Changes involving the modification of the replication style of a group are discussed in Section 4.2.2.

4.2 OGS+

OGS+ is an extension of OGS (Object Group Service) [4], a Group Communication service that implements the service approach (discussed in Section 3.1) for the incorporation of Group Communication in CORBA and that provides means to create and maintain CORBA object groups using only CORBA standardized constructs. OGS+ was built upon OGS version 0.8b1, which was implemented with *Java* and Inprise´s VisiBroker 3.2 CORBA ORB. However, before we made extensions to OGS we upgraded it to use VisiBroker 4.0, which is based on CORBA 2.3 specification and, thus, provides a much more elaborate Object Adapter, called *Portable Object Adapter* (POA).

OGS is composed of four main components: *Group Service*, *Consensus Service*, *Monitoring Service* and *Messaging Service*. The *Group Service* allows the management of object groups and guarantees a consistent state between group members. It also provides means to send multicast invocations to object groups with varying degrees of ordering and reliability. The *Consensus Service* solves some consensus-related problems in OGS, such as group membership and total ordering of messages. The *Monitoring Service* provides mechanisms to detect faulty objects and to disseminate fault event notifications. Finally, the *Messaging Service* supports different kinds of reliability, ordering and synchrony when sending messages through OGS.

In OGS, the management of object groups on the server side is implemented through *Group Administrator* objects, which are associated with each object replica in the group. In addition, these objects interact with each other to execute consensus and voting protocols. It is within *Group Administrator* objects that OGS implements active replication; the single replication style supported. 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 (i.e. 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 knowing how many replicas exists and where they are executing. Thus, the *Group Accessor* is the one which sends multicasts to the *Group Administrator* objects and waits for replies from them.

The protocol used to maintain a consistent state between object replicas when active replication is used can generate a communication overload and a considerable delay when processing requests sent to a group. This happens because every request must be delivered and processed in the same order by each group member. However, for many applications it is enough that only a small subset of the object replicas within a group process the requests sent to the group. In order to attend also these requirements we have created OGS+, an extension of OGS that provides replication styles other than active replication and that supports dynamic switching between them.

4.2.1 Replication Styles

In addition to the ACTIVE replication style originally implemented in OGS, OGS+ also supports three replication styles specified in FT-CORBA, namely STATELESS, COLD_PASSIVE and WARM_PASSIVE (see Section 2.1). Additionally, OGS+ also implements a variant of the VIRTUAL_PRIMARY_COPY replication style [13], which can be seen as a generalization of the WARM_PASSIVE replication style.

In order to implement these replication styles two new objects were defined: *Extended Group Administrator* and *Extended Group Accessor*. *Extended Group Administrator* objects extend *Group Administrator* objects and implement the new replication styles and a protocol to dynamically switch between them. *Extended Group Accessor* objects hold a precise information concerning the replication style currently used by the object groups to which they connect and allow multicast communication with the *Extended Group Administrator* objects.

In the VIRTUAL_PRIMARY_COPY replication style implemented in OGS+ one or more replicas are designated as primaries and form a *consistency island*, or simply an *island*, where ACTIVE replication style is used. Thus, from the viewpoint of the application, the set of replicas within the island can be considered as if it were a single virtual primary replica. The other replicas within the group are backups of the virtual primary replica (i.e. the set of primaries). Moreover, in the VIRTUAL_PRIMARY_COPY replication style the interaction between the virtual primary replica and the backups is identical as occurs in WARM_PASSIVE replication.

The main advantage of using this replication style when compared to WARM_PASSIVE replication is a reduced time to execute the failover mechanism, i.e. to switch a backup replica into a primary. The reason is that when a primary replica fails, other primaries still exist in the island and can continue receiving and processing requests as usual. When compared to ACTIVE replication, VIRTUAL_PRIMARY_COPY can be particularly faster when used for large groups with small islands (about 3 or 4 members), because it avoids the costly execution of the consensus protocol between all the group members, as required with ACTIVE replication.

4.2.2 Dynamic Modification of the Replication Style

In order to allow a more flexible configuration of applications based on replicated objects, OGS+ provides means to dynamically modify the replication style associated to an object group. In this way, the applications can better adapt themselves to the available network resources, such as the number of hosts or the network bandwidth. Thus, OGS+ allows the system administrator to have a very accurate control over services whose use frequency (e.g. the number of requests per unit of time) shows peaks of highs and lows.

Figure 10 shows an example that describes how the protocol for dynamic change of the replication style associated to an object group works. It shows the steps performed during the modification of the replication style of a group composed of two replicas (located at hosts "A" and "B") from ACTIVE to WARM_PASSIVE. Let´s assume that after this modification the primary replica of the group will be the one located at host "A" and that during the modification of the replication style the object group is being accessed by a client.

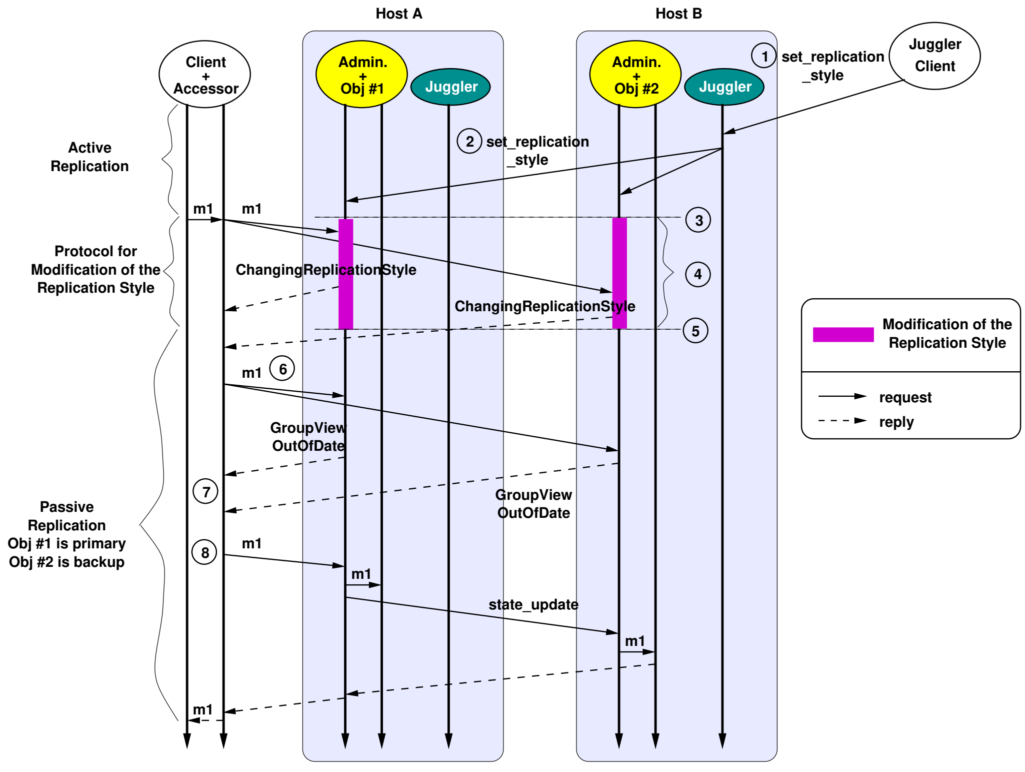

Figure 10: Dynamic modification of the replication style of an object group

In order to change the replication style of a group a client of Juggler/OGS+ obtains a reference to a *ReplicationManager* object and invokes **set_replication_style** operation (step 1), which is just sent via multicast to the *Extended Group Administrators* associated to the group members (2). The *Extended Group Administrators* then block the processing of new requests (3). While the replication style is being changed any request sent by clients of the group (in fact, by the *Extended Group Accessors*) is replied with a *ChangingReplicationStyle* exception (4). After receiving such reply, an *Extended Group Accessor* enters a loop and tries to reissue the request after a fixed time period (e.g. 1 second).

After updating their internal structures that store the replication style of the group, the *Extended Group Administrators* unblock the group so that it can receive and process new requests (5). At this point, the replica located at host "A" becomes the primary replica and the one located at host "B" becomes the backup. At the client side, the *Extended Group Accessor* object reissue the multicast to the group (6), which is promptly replied with a *GroupViewOutOfDate* exception (7), which informs the *Extended Group Accessor* that the group view it has is out of date. This exception also carries information concerning the new replication style associated to the group. Finally, the *Extended Group Accessor* updates its view of the group and reissues the multicast once more (8). This time, however, it only sends the request to the primary replica of the group. The request is then processed, the state of the backup replica is updated and the reply is returned to the client.

5 Tests and Results

A fully functional prototype of Juggler/OGS+ was developed using VisiBroker for Java 4.0. For test purposes, we have implemented a small banking application composed of replicated objects. These objects implement the `Account` interface, which specifies operations to deposit and withdraw money from a bank account, as well as an operation to get the current balance, which is, in fact, replicated between the object replicas. Tests were performed on a local 10Mbit Ethernet network interconnecting 9 Sun SPARCstations 4, running Solaris 7, shared also by other users. The tests were run with the TCP_NODELAY option set, which disables buffering and forces the immediate sending of requests.

In the first test (Figure 11) we measured the total time spent by Juggler/OGS+ to create object groups with 1 to 9 members. The results shown in the graph represent the average values obtained from 25 runs of the test for each group size. From the graph we can see that as the number of replicas in the group increases, the time needed to create the group grows almost exponentially. For instance, groups with 3 and 6 replicas were created in approximately 3.7s and 15.4s respectively, whereas in average a group with 9 replicas took 35.1s to be created. This happens mainly because the use of the CORBA Naming Service by OGS, and also by OGS+, during the creation of object replicas is done under mutual exclusion, i.e. object replicas are sequentially instantiated so as to guarantee that during the creation of the group the replicas have a consistent group view.

In the graph of Figure 11, the values represented by the upper line in the graph are accumulated, i.e. they represent the sum of the wait time in the CORBA Naming Service (CORBA NS) plus the time spent to create the object replicas. The lower line in the graph represents the time the last object replica spent by waiting 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 (about 80% of the time), and, thus, the actual overhead introduced by Juggler/OGS+'s creation protocol is relatively small.

The second test performed was to measure the time needed by Juggler/OGS+ to recover failed object replicas. In order to run this test, we created groups with 1 to 9 members and simulated the failure of one of these members. During the execution of this test the time necessary to detect the failure was not computed, since it essencially depends on monitoring parameteres defined in the OGS+ Monitoring Service and can be adjusted to arbitrary values in each host where Juggler/OGS+ is running. As in the previous graph, the values shown in the lines of the graph are accumulated and the results represent the average values obtained from 25 runs of the test for each group size.

The failure recovery process in Juggler/OGS+ basically consists of three steps: (1) notification concerning the failure is sent to the *MANAGERS* group; (2) choice of the *GroupManager* responsible

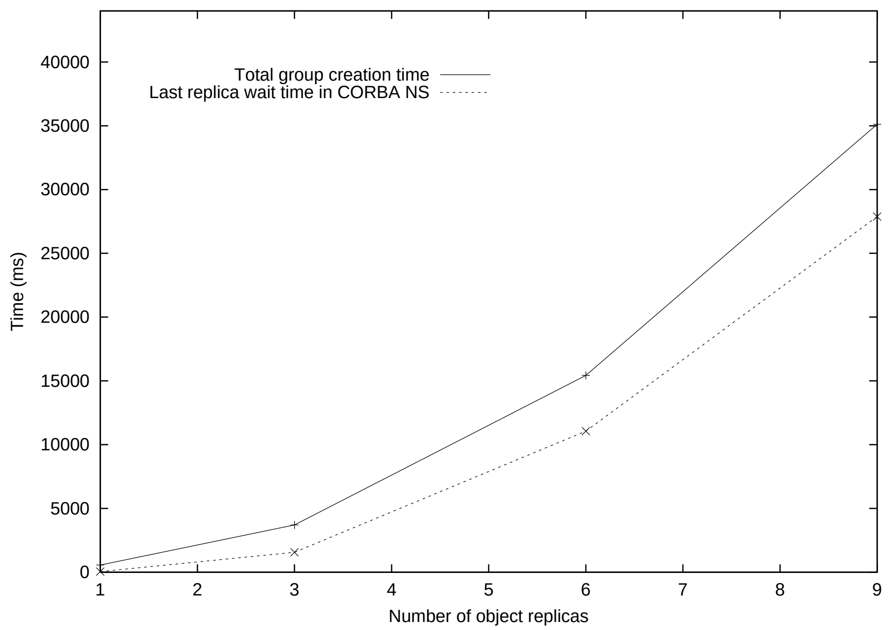

Figure 11: Time spent to create object groups through Juggler/OGS+

to carry on the failure recovery; and (3) creation of a new group member. Figure 12 illustrates the time spent to perform each of these steps.

For groups composed of 3 replicas, the time necessary to send a notification concerning the failure (step 1) was about 234ms. Steps (2) and (3) consumed, on the average, 115ms and 1.29s respectively, which makes up a total recovery time of about 1.64s. For groups with 6 and 9 replicas the total time spent to recover a failed object was about 3.65s and 6.04s, respectively. From the graph we can see that the creation of a single object replica grows in a non-linear way. This is mainly due to the fact that the creation of an object through Juggler/OGS+ involves the exchange of several multicast messages and that the reliable multicast protocol implemented in OGS spends an exponential time to perform this task, as described in [4].

Another test with our prototype aimed at measuring the time spent by an application client to issue and receive a response for an atomic multicast sent to an object group. This test was performed for groups composed of 3 members being concurrently accessed by 1, 5, 10 and 15 clients. Moreover, the test was done for groups with ACTIVE and WARM_PASSIVE replication styles. The results of this test are shown in Figure 13. The values shown in the graph represent average values for sending a few hundred multicasts by each client set.

As can be seen from Figure 13, the time spent to send an atomic multicast to groups which adopt ACTIVE replication was on the average between 50% and 60% higher than when WARM_PASSIVE replication style was used. The difference between the times measured for ACTIVE and WARM_PASSIVE replication can be much higher if we make a small improvement to the protocol that implements the WARM_PASSIVE replication style. This improvement consists of sending a reply to the client as soon as the operation has been processed by the primary replica, instead of waiting for the acknowledgements of the state updates from the backup replicas before sending the reply to the client.

A fourth test measured the time to issue (and receive the reply of) an atomic multicast sent by a client while the replication style associated to a group is dynamically modified. In this test we measured the time to modify from ACTIVE to WARM_PASSIVE replication styles, and vice-versa. To perform this test a group with 3 members was created and its replication style was alternately switched between ACTIVE and WARM_PASSIVE replication.

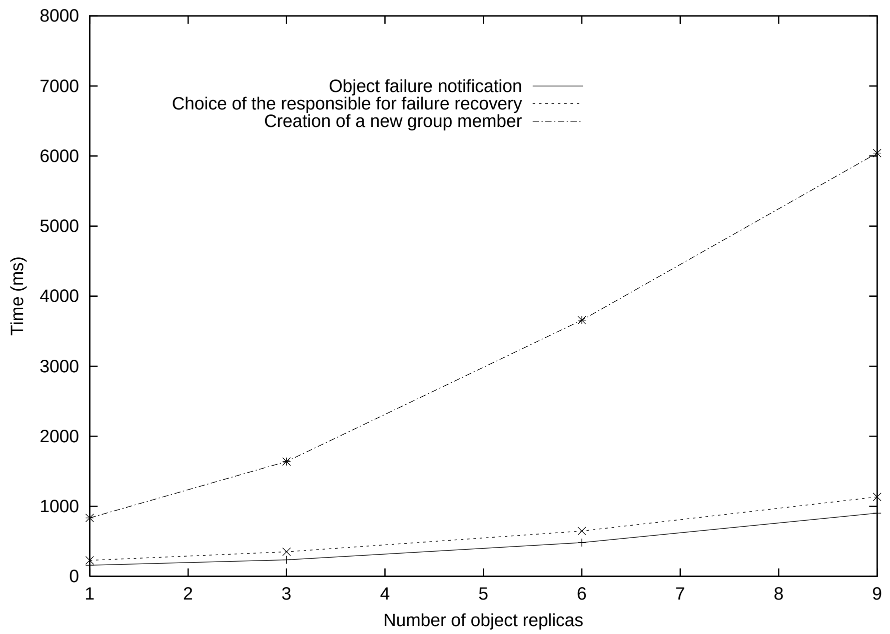

Figure 12: Time spent during the recovery of a failed object replica

During the modification from ACTIVE to WARM_PASSIVE replication style, while the group was accessed by 1 and 5 clients, the time spent was on the average 163ms and 2.02s, respectively. The opposite operation consumed 268ms and 3.19s, respectively, for 1 and 5 clients. Thus, our measurements indicate that the change from WARM_PASSIVE to ACTIVE took about 50% more time than when changing from ACTIVE to WARM_PASSIVE replication style.

For groups accessed by 10 and 15 clients, the modification from ACTIVE to WARM_PASSIVE replication spent, respectively, 2.16s and 6.71s. However, when we performed the opposite operation the information concerning the replication style, which is stored in the *Extended Group Administrators*, became inconsistent due to the large number of concurrent requests sent to group, thus blocking the group for the reception of new requests. This problem occurs due to a design decision made in the OGS Consensus Service. In OGS a consensus is reached even if there is only the single majority of the propositions of the participants in the consensus protocol. Thus, if the primary replica of the group becomes flooded with client requests, a request concerning the replication style modification may be delivered first to the backup replicas and later to the primary replica. So, it can happen that some *Extended Group Administrators* store out of date information concerning the replication style associated to the group, hence preventing the group from processing new requests.

6 Conclusion

Support for fault tolerance is an usual requirement of many 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 small set of mechanisms for managing object groups.

Through our work we aimed at providing high-level abstractions that ease the management of fault tolerant CORBA applications. Juggler/OGS+ supports monitoring and failure recovery of object replicas, availability management and specification of several management policies for fault tolerant applications. The main contributions of Juggler/OGS+ are its set of high-level interfaces to create and manage CORBA object groups and an Object Group Service that offers a rich set of fault tolerance properties for CORBA replicated objects. Another interesting feature of our service is that

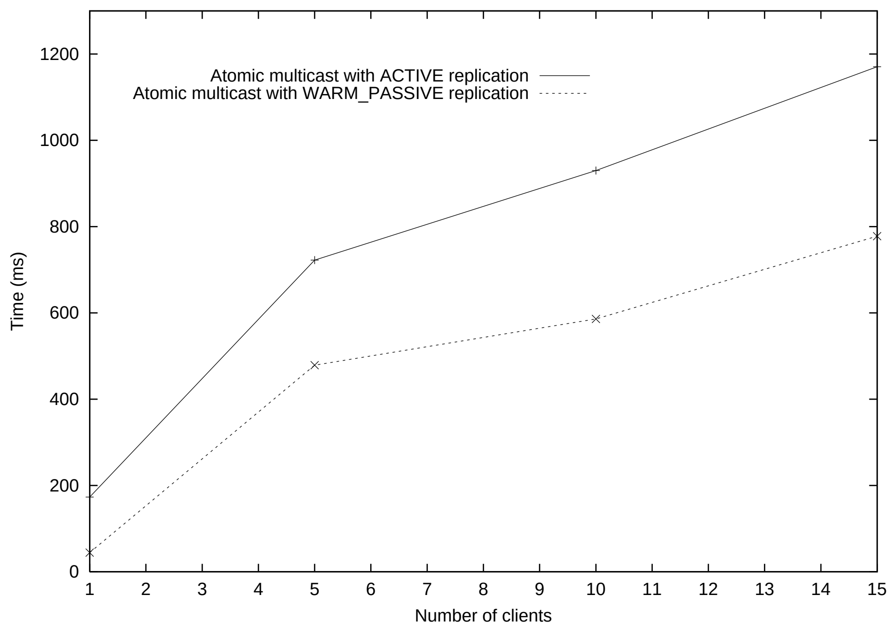

Figure 13: Time spent to send an atomic multicast to an object group

the Juggler/OGS+ layers are entirely portable and interoperable with different CORBA ORBs.

In our work we adopted some ideas present in related works. Like Piranha, Juggler/OGS+ replicates the information concerning object groups managed by the service. The flexibility of specifying different management policies for groups of object replicas came from Proteus. In addition, Juggler also implements a full set of services, such as a generic Object Factory, a fault notification service and mechanisms for checkpointing and logging, which follow the guidelines specified in the FT-CORBA standard.

One of the main differences between Juggler/OGS+ and FT-CORBA is that our fault tolerance infrastructure implements protocols that support both active and primary-backup replication, as well as a variant of virtual-primary-copy replication and a means to dynamically switch between them, which is not specified in FT-CORBA. In our project we do not make any provision to support fault analyzers. Finally, a major difference between the architecture proposed by Juggler/OGS+ and that of FT-CORBA is that we do not implement mechanisms for creating logs in stable storage, thus supporting only a very simple recovery mechanism. At last, we should note that the main goal when developing Juggler/OGS+ is to create a flexible management service for CORBA object groups that can be run across different CORBA ORBs and not a generic fault tolerance infrastructure such as the one standardized by OMG.

We have implemented and tested a fully functional prototype of Juggler/OGS+. We have evaluated its performance and identified the major overheads associated with the implemented protocols. In general, the performance of Juggler/OGS+ is acceptable and the service is suitable to applications that do not need high-end performance for multicast communication with object groups and that do not have real-time processing requirements.

It is also important to note that the overhead of the group management service offered by Juggler will be noticed only by the applications that use it, without affecting the overall performance of other applications. Moreover, Juggler introduces only a little overhead to fault tolerant applications since it only has effect during the initial creation of object groups and when the semantics of the groups are changed. In fact, during most of the time groups are accessed directly through OGS+.

By the time being we are implementing features that allow the safe use of Juggler/OGS+ when accessed by many clients concurrently and developing a Graphical User´s Interface that shall provide a friendly user interface for the system administrator. The complete Juggler/OGS+ software is free, open source and can be downloaded from the URL *http://www.ime.usp.br/˜sidam/components/juggler/*.

References

[1] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. *2nd International Conference on Software Engineering*, pages 562–570, 1976.

[2] 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*, pages 245–253, West Lafayette, Indiana, USA, October 1998. IEEE.

[3] M. A. M. de Moura. Uma Infra-Estrutura para o Gerenciamento de Aplicações CORBA Tolerantes a Falhas. Master's thesis, Universidade de São Paulo, Brasil, Fevereiro 2001. In Portuguese.

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

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

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

[7] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical Report formal/00-10-07, Object Management Group, 2000. Version 2.4.

[8] Object Management Group. CORBAservices: Notification Service Specification. Technical Report formal/00-06-20, Object Management Group, June 2000.

[9] Object Management Group. Fault Tolerance FTF Report. Technical Report ptc/00-12-06, Object Management Group, December 2000.

[10] Object Management Group. Fault Tolerant CORBA Specification, V1.0. Technical report, April 2000. ptc/2000-04-04.

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

[12] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[13] R. Lenz. The V̈irtual-Primary-Copy ApproachC̈ompared to other Approaches with Weak Consistent Data Replication. In *II Workshop on Mobility and Replication, ECOOP'96*, Linz, Austria, July 1996.

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

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

[16] 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.

[17] 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.

[18] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards High-performance Fault Tolerant CORBA. In *Proceedings of the 2nd Distributed Applications and Objects (DOA) Conferece*, Antwerp, Belgium, September 2000.

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

[20] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[21] 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.